Računske vježbe 7

Programabilni uređaji i objektno orijentisano programiranje

1. Realizovati šablonsku funkciju **arrange** kojom se od dva uređena neopadajuća niza formira treći, na isti način uređen niz. Realizovati glavni program koji primjenjuje ovu funkciju nad nizovima cijelih brojeva i nizovima tačaka, pri čemu koordinate tačaka mogu biti cijeli ili realni brojevi. Klasu **Point** realizovati kao šablonsku klasu kako bi koordinate mogle biti traženog tipa.

```
#include <iostream>
#include <cmath>

using namespace std;

template<typename T>
void arrange(T *first, int nFirst, T *second, int nSecond, T *result)
{   // brojaci f za first, s za second, r za result...
    for(int f = 0, s = 0, r = 0; f < nFirst || s < nSecond; r++)
        if(f == nFirst)
            result[r] = second[s++];
        else if(s == nSecond)
            result[r] = first[f++];
        else
        {
            if(first[f] < second[s])
                result[r] = first[f++];
            else
                result[r] = second[s++];
        }
}

template<typename S>
class Point
{
private:
    S x;
    S y;
public:
    Point(S = 0, S = 0);
    bool operator<(const Point&) const;
    void print()
    {
        cout << "(" << x << "," << y << ")" <<endl;
    }
};

template<typename S>
Point<S>::Point(S x, S y): x(x), y(y) {}

template<typename S>
bool Point<S>::operator<(const Point& point) const
```

```
{
    return (pow(x, 2) + pow(y, 2)) < (pow(point.x, 2) + pow(point.y, 2));
}

int main()
{
    int arr1[]= {1, 2, 5, 7, 9};
    int arr2[]= {3, 4, 7, 8};
    int *arrResult = new int[9];
    //primjer niza cjelobrojnih vrijednosti, tip se zakljucuje!
    arrange(arr1, 5, arr2, 4, arrResult);

    cout << "Uredjeni niz je:" << endl;
    for(int i = 0; i < 9; i++) cout << " " << arrResult[i];
    cout << endl;

    delete []arrResult; //oslobadjamo memoriju
    //primjer niza tipa Point cije su koordinate tipa int
    cout << "Koliko ima tacaka prvi niz?" << endl;
    int nFirst;
    cin >> nFirst;
    Point<int> *first;
    first = new Point<int>[nFirst];

    cout << "Unesite niz tacaka sa cjelobrojnim koordinatama" << endl;
    int x, y;
    for(int i = 0; i < nFirst; i++)
    {
        cin >> x >> y;
        first[i] = Point<int>(x, y);
    }

    cout << "Koliko ima tacaka drugi niz?" << endl;
    int nSecond;
    cin >> nSecond;
    Point<int> *second;
    second = new Point<int>[nSecond];

    cout << "Unesite niz tacaka sa cjelobrojnim koordinatama" << endl;
    for(int i = 0; i < nSecond; i++)
    {
        cin >> x >> y;
        second[i] = Point<int>(x, y);
    }

    int nResult;
    nResult = nSecond + nFirst; //racunamo duzinu rezultujuceg niza

    Point<int> *result;
    result = new Point<int>[nResult];

    arrange(first, nFirst, second, nSecond, result);

    cout << "Uredjeni niz je:" << endl;
    for(int i = 0; i < nResult; i++)
    {
        cout << " ";
        result[i].print();
    }
```

```
    cout << endl;

    delete [] first;
    delete [] second;
    delete [] result;
    //tacke cije su koordinate tipa double
    Point<double> point1(2.3, 4.6), point2(2.5, 6.8);

    cout << "Tacka ima koordinate: ";
    (point1 < point2) ? point1.print() : point2.print();
}
```

Kada smo na jednom od uvodnih časova naglašavali prednosti programskog jezika `C++` nad programskim jezikom `C` pomenuli smo pojam preklapanja funkcija. Ako za primjer uzmemo problem pronalaženja većeg od dva podatka (bilo kojeg tipa) to bi značilo da bismo za svaki tip podataka pisali po jednu funkciju. Istine radi, u programskom jeziku `C` ove funkcije nisu mogle imati isto ime te smo preklapanje funkcija smatrali značajnim iskorakom. Međutim, u primjeru pronalaženja većeg od dva podatka, razlike između ovih funkcija svode se na sistematsko zamjenjivanje oznake tipa unutar cijele definicije funkcije:

```
int max (int a, int b) { return a > b ? a: b; }
double max (double a, double b) { return a > b ? a: b; }
char max (char a, char b) { return a > b ? a: b; }
```

što predstavlja rutinski posao koji se može automatizovati odnosno generalizovati. Jezik `C++` omogućava nam definisanje šablona (engl. *template*) pomoću kojih opisujemo datu obradu u vidu opšteg slučaja bez navođenja konkretnih tipova podataka. Pomoću ovako definisanih šablona se automatski generišu konkretne funkcije za konkretne tipove podataka. Ovo generisanje se vrši po potrebi, ne prave se istovremeno sve moguće kombinacije za sve raspoložive tipove. Šabloni se mogu definisati za funkcije, klase i još ponešto. Opšti oblik šablona za generičke stvari bi bio:

```
template <parametar, parametar, ... , parametar> stvar
```

gdje `stvar` predstavlja definiciju ili deklaraciju funkcije, klase itd. čiji se šablon definiše. Šabloni nemaju oznaku kraja i završavaju se tamo gdje se završava opisivana stvar. Parametri šablona se pišu unutar `<>` zagrada i mogu da označavaju kako tipove tako i konstante. Opšti oblik parametra je:

```
class Ime
typename Ime
tip Ime
```

gdje ključne riječi `class` i `typename` označavaju tipovne parametre dok sa `tip` predstavljamo netipovne parametre koji su zapravo konstante navedenog tipa, recimo `int k`. Ključna riječ `typename` nam govori da se može raditi o bilo kojem tipu dok sa `class` naglašavamo one parametre šablona koji moraju biti klase. Važno je znati da ovo nije nešto što nameće sam programski jezik već standard kojeg ćemo se pridržavati. Naime, inicijalna verzija programskog jezika `C++` nije dodala novu ključnu riječ kojom bi se označavali tipovni parametri već je samo upotrijebila ključnu riječ `class` što je stvorilo veliku konfuziju jer tipovni parametar može biti bilo kojeg tipa, ne radi se o klasi. Zbog toga je naknadno uvedena nova ključna riječ `typename`. Kako se stari programi ne bi pokvarili ostavljena je mogućnost i da se koristi `class` kojoj smo našli praktičnu namjenu. Imena parametara šablona obično se pišu velikim početnim slovom. Najčešće se koristi slovo T (type) ili njemu u engleskom alfabetu susjedna slova S i U. Dakle, radi se o standardu, ne i o obavezi. Pa, kako bismo napisali šablon za jednu funkciju? Sjetimo se našeg pokaznog primjera za određivanje većeg od dva podatka:

```
template<typename T> T max (T a, T b) { return a > b ? a : b; }
```

gdje sa T označavamo tipove argumenata, ali i tip rezultata funkcije! Ovaj šablon se može koristiti i za klase, potrebno je ,,samo" preklopiti operator veće za one klase od interesa. Funkcija se generiše na osnovu šablona po potrebi, odnosno:

```
int a = max<int>(1, 2);
```

gdje između <> prosljeđujemo tip od interesa. Prilikom generisanja, tip parametra se može i zaključiti:

```
int a = max(1, 2);
double b = max(1, 2.0) // greska, tipovi se ne uklapaju (int i double)!
```

Slično funkcijama, šablonima se mogu definisati i generičke klase:

```
template<parametri> class Ime { clanovi };
```

pa smo u slučaju naše klase imali:

```
template<typename S>
class Point
{
private:
    S x;
    S y;
public:
    Point(S = 0, S = 0);
};
```

i uočavamo da nam S predstavlja tip koordinata koji koristimo u njihovoj deklaraciji, ali naravno i u konstruktoru i gdje god nam to u klasi treba. Kada želimo da definišemo neku metodu ili konstruktor koji je u tijelu klase (odnosno unutar šablona) samo deklarisan, koristićemo operator dosega, ali moramo obavezno naglasiti da se radi o šablonu:

```
template<typename S>
Point<S>::Point(S x, S y): x(x), y(y) {}
```

i još jednom skrenimo pažnju da se šablon završava tamo gdje se završava opisivana stvar (u ovom slučaju konstruktor) te da slovo S koje predstavlja tip parametra možemo koristiti u prvom narednom šablonu, a da smo u deklaraciji klase mogli koristiti neko drugo.