Podstawy programowania: Laboratorium nr 10
Szablony.

2017-2018

mgr inż. Przemysław Walkowiak
dr inż. Michał Ciesielczyk

Instrukcja

W czasie pisania programu pamiętaj o:

1. dbaniu o czytelność kodu (odpowiednie formatowanie kodu, nazewnictwo zmiennych adekwatne do ich znaczenia, komentarze),
2. dbaniu o czytelność interfejsu z użytkownikiem (w sposób jawny pytaj użytkownika jakie dane ma podać oraz opisuj wyniki, które zwracasz),
3. przed fragmentem implementującym poszczególne zadania umieść komentarz: `/*Zadanie X */` oraz wypisz na ekranie analogiczny komunikat (X jest numerem zadania): `std::cout << "Zadanie X"<< std::endl;`,
4. umieszczeniu wszystkich rozwiązań w jednym pliku, chyba, że w poleceniu napisano inaczej.
5. w zadaniach wymagających udzielenia komentarza bądź odpowiedzi, należy umieścić go w kodzie programu (np. w postaci komentarza albo wydrukować na ekranie).

Wprowadzenie

Szablony

Szablony (ang. templates) są kolejnym mechanizmem wprowadzonym w języku C++ pozwalającym zmniejszyć częstość pojawianie się duplikatów w kodzie. Przykładowo załóżmy, że mamy zdefiniowane trzy macierze o następujących typach:

```
std::vector<std::vector<double>> m_double;
std::vector<std::vector<int>> m_int;
std::vector<std::vector<complex>> m_complex;
```

oraz chcemy zaimplementować podstawowe operacje na tych macierzach takie jak mnożenie, dodawanie, odejmowanie. W klasycznym podejściu konieczne jest zaimplementowanie oddzielnego zestawu funkcji dla każdego typu:

```
std::vector<std::vector<double>> add(
                            const std::vector<std::vector<double>>& A,
                            const std::vector<std::vector<double>>& B);

std::vector<std::vector<int>> add(
                            const std::vector<std::vector<int>>& A,
                            const std::vector<std::vector<int>>& B);

std::vector<std::vector<complex>> add(
                            const std::vector<std::vector<complex>>& A,
                            const std::vector<std::vector<complex>>& B);
```

pomimo tego, że algorytm i implementacja są praktycznie identyczne. Aby uniknąć takiego powtarzania można wykorzystać szablony, które zamienią fragment implementacji we wzór, który będzie modyfikowany w zależności od potrzeb programisty.

Szablonem w C++ może zostać dowolna funkcja, metoda, struktura czy klasa poprzez dodanie słowa kluczowego `template<T>` przed deklaracją/definicją odpowiedniego elementu. Na przykład dla powyższych funkcji wykorzystanie wygląda następująco:

```
template <typename T>
std::vector<std::vector<T>> add(
                            const std::vector<std::vector<T>>& A,
                            const std::vector<std::vector<T>>& B);
```

a wykorzystanie następująco:

```
add(m_double, m_double);
add(m_int, m_int);
add(m_complex, m_complex);
```

Wywołanie funkcji dodaj wygląda identycznie jak w przypadku klasycznym, jednakże główną różnicą jest to, że nie jest konieczne przeciążanie i implementowanie funkcji dla każdego możliwego typu. Kompilator sam zadba o wygenerowanie odpowiedniej implementacji na podstawie szablonu.

Dla struktur wykorzystanie szablonów wygląda następująco:

```
template <typename T>
struct Point {
    T x;
    T y;

    Point<T> operator+(Point<T> &p);
}

Point<double> punkt_double;
Point<int> punkt_int;


template <typename T>
struct BstNode {
    int value;
    std::unique_ptr<BstNode<T>> left;
    std::unique_ptr<BstNode<T>> right;

    BstNode(T value);
};

std::unique_ptr<BstNode<int>> bst;
std::unique_ptr<BstNode<float>> bst;
std::unique_ptr<BstNode<Student>> bst;
```

Ponieważ kompilator w momencie generowania implementacji funkcji, struktury musi znać wiedzieć wszystko danym szablonie (musi znać również ciało funkcji szablonowej), przez co wszystkie szablony powinny być umieszczone w pliku nagłówkowym modułu.

Wykorzystanie typów szablonowych w funkcjach

Załóżmy, że mamy dany szablon funkcji

```
template <typename T>
std::vector<std::vector<T>> createMatrix(int m, int n) {
\\ ...
}
```

Zadaniem powyższej funkcji ma być utworzenie i zwrócenie macierzy o wymiarach $m \times n$. Gdybyśmy mieli do czynienia z konkretnym typem np. `int` to macierz mogłaby zostać utworzona w następujący sposób:

```
    std::vector<std::vector<int>> A(m);
    for (unsigned int i = 0; i < m; i++) {
        A[i] = std::vector<int>(n);
    }
```

Dla typu szablonowego zapis jest analogiczny:

```
    std::vector<std::vector<T>> A(m);
    for (unsigned int i = 0; i < m; i++) {
        A[i] = std::vector<T>(n);
    }
```

W momencie procesu specjalizacji szablonu `T` zostanie zastąpione odpowiednim typem wskazanym przez użytkownika/programistę. Kompletna funkcja będzie miała następującą postać:

```
template <typename T>
std::vector<std::vector<T>> createMatrix(int m, int n) {
    std::vector<std::vector<T>> A(m);
    for (unsigned int i = 0; i < m; i++) {
        A[i] = std::vector<T>(n);
    }
    return A;
}
```

Ponieważ wśród jawnych argumentów funkcji nie pojawia się żaden z typem `T` w związku z tym trzeba wprost określić ten typ w czasie wywołania, tzn.:

```
std::vector<std::vector<int>> A1 = createMatrix<int>(5,5);
std::vector<std::vector<float>> A2 = createMatrix<float>(5,5);
std::vector<std::vector<complex>> A3 = createMatrix<complex>(5,5);
```

W momencie kompilacji na podstawie zostaną wygenerowane następujące funkcje:

```
std::vector<std::vector<int>> createMatrix(int m, int n) {
    std::vector<std::vector<int>> A(m);
    for (unsigned int i = 0; i < m; i++) {
        A[i] = std::vector<int>(n);
    }
    return A;
}
std::vector<std::vector<float>> createMatrix(int m, int n) {
    std::vector<std::vector<float>> A(m);
    for (unsigned int i = 0; i < m; i++) {
        A[i] = std::vector<float>(n);
    }
    return A;
}
std::vector<std::vector<complex>> createMatrix(int m, int n) {
    std::vector<std::vector<complex>> A(m);
    for (unsigned int i = 0; i < m; i++) {
        A[i] = std::vector<complex>(n);
    }
    return A;
}
```

Oczywiście trzeba tutaj zaznaczyć, że gdybyśmy ręcznie powyższe funkcje zaimplementowali w ten sposób kompilacja by się nie powiodła – mamy trzy funkcje o tej samej nazwie i liście argumentów.

Kolejny przypadek wykorzystania szablonów obejmuje występowanie typu szablonowego na liście argumentów funkcji, np.:

```
template <typename T>
std::vector<std::vector<T>> copy(const std::vector<std::vector<T>>& matrix) {
    unsigned int m = matrix.size();
    unsigned int n = matrix[0].size();

    std::vector<std::vector<T>> matrixCopy = createMatrix(m, n);
    for (unsigned int i = 0; i < m; i++) {
        for (unsigned int j = 0; j < n; j++) {
            matrixCopy[i][j] = matrix[i][j];
    }
    return matrixCopy;
}
```

Przykład programu wykorzystujący tę funkcję może wyglądać następująco:

```
auto A = createMatrix<int>(5, 6);
auto B = copy(A);
```

Jak można zauważyć w wywołaniu funkcji `copy` nie potrzeba już jawnie określać typu szablonu.

Wykorzystanie typów szablonowych w strukturach

W grafice komputerowej wykorzystuje się bardzo często struktury reprezentujące punkt na płaszczyźnie (lub w przestrzeni). Przykładowa implementacja została już zrobiona na poprzednich zajęciach. Jednakże w zależności od konkretnego zastosowania powinno wybierać sie reprezentację o typie bazowym określonej precyzji. Np. w grafice 3D wykorzystuje się zazwyczaj zakres liczbowy od 0 do 1 (liczba zmiennoprzecinkowa), natomiast na płaszczyźnie wykorzystuje się liczby całkowite (liczba pikseli). Ponieważ niezależnie od precyzji typu bazowego punktu zachowuje się on zawsze tak samo, idealnym rozwiązaniem jest wykorzystanie szablonów.

Struktury szablonowe deklaruje się w sposób analogiczny jak funkcje szablonowe:

```
template <typename T>
struct Point {
    T x;
    T y;
    Color color;
    Point (T v) {
        x = y = v;
    }
    Point () {
        x = T();
        y = T();
    }
}
```

Analizują przykład można zauważyć, że typ szablonowy może zostać zastosowany zarówno w typach pól struktury (`x, y`) jak i w jej metodach czy konstruktorach. Konstruktor domyślny w powyższym przykładzie ma za zadanie wypełnić pola `x, y` wartościami domyślnymi. Moze zostać to zrealizowane poprzez wywołanie konstruktora domyślnego typu szablonowego `T()`. Co jest odpowiednikiem np.: `int()` lub `float()` - czyli `x, y` zostaną zainicjalizowane wartościami domyślnymi odpowiednich typu a dokładniej wartością 0.

Wykorzystanie struktury szablonowej wygląda następująco:

```
Point<int> point_int(5);
Point<float> point_float(2);
Point<float> point2_float();
```

Dostęp do pół struktury się nie zmienił:

```
point_int.x = 2;
point_float.x =2.4;
point2_float.y = 1.3;
```

Należy pamiętać, że wszystkie funkcje i struktury szablonowe muszą być zaimplementowane w plikach nagłówkowych!

Dodatkowe informacje:

- szablony - http://www.cplusplus.com/doc/tutorial/templates/

Zadania

Zadanie 1

Z wykorzystaniem szablonów oraz zadań z poprzednich laboratoriów zaimplementuj obsługę liczb zespolonych, gdzie precyzja reprezentacji części rzeczywistej i urojonej jest określona przez użytkownika/programistę biblioteki. Wykorzystując szablonową wersję implementacji liczb zespolonych napisz program, który je wykorzystuje. Użyj typów bazowych: **int**, **float**, **double**.

Zadanie 2

Z wykorzystaniem szablonów oraz zadań z poprzednich laboratoriów zaimplementuj operacje dodawania, odejmowania i mnożenia skalarnego dwóch wektorów o długości n (tablice jednowymiarowe) oraz wyświetlania na standardowym wyjściu. Następnie przetestuj działanie dla par wektorów o następujących typach: **int**, **float**, `complex`. Wektory możesz wypełnić wartościami losowymi. Wykorzystaj zaimplementowana wcześniej bibliotekę liczb zespolonych.

Zadanie 3

Przygotuj operacje na macierzach z poprzednich laboratoriów (dodawanie, odejmowanie, mnożenie macierzy, wyświetlanie macierzy), a następnie z wykorzystaniem szablonów uogólnij funkcje ze względu na typ komórki macierzy. Napisz kilka przykładowych fragmentów kodu wykorzystując jako typ podstawowy typy: **int**, **double**, `complex<int>` oraz `complex<float>`. Wartości macierzy możesz wylosować.

Implementacje umieść w oddzielnym pliku nagłówkowym – np. *matrix_ operations.hpp*.

Zadanie 4

Wykorzystując szablony zaimplementuj uogólnioną obsługę drzew typu BST z poprzednich laboratoriów. Przetestuj swoją implementację dla typu **float** oraz **unsigned int**.

Na następne zajęcia

- Wyrażenia lambda: `http://en.cppreference.com/w/cpp/language/lambda`.