Лекція 18.
Шаблони функцій та класів

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

План на сьогодні

1. Шаблонні функції
2. Шаблонні класи

Шаблони функцій

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Шаблони

- Шаблон — це простий, але дуже потужний інструмент у C++. Основна ідея полягає в тому, щоб **передавати тип даних як параметр**, що дозволяє **уникнути** написання **однакового коду для різних типів даних**.
- C++ додає два нові ключові слова для підтримки шаблонів: template і typename. Друге ключове слово завжди можна замінити на class.

- Шаблони дозволяють задавати узагальнені, в розумінні довільного вибору типів, визначення класів та функцій
- Надалі ці визначення слугують компілятору основою для створення класів та функцій, що використовують конкретні типи даних.
- Шаблон класу (class template) визначає дані та операції потенційно необмеженої множини родинних типів
- Шаблон функції (function template) визначає необмежену множину родинних функцій

Синтаксис шаблонної функції

template < *template-parameter-list* > *declaration*

- ***template-parameter***
  - ***Параметр-тип***
    - typename *identifier [= typename]*
    - class *identifier [= typename]*
- ***declaration:***
  - Оголошення **class** *або* **struct**
  - Оголошення **функції**
- ***Шаблонний тип***
  - *template< template-parameter-list > class [identifier] [= name]*

Синтаксис шаблонної функції

- template <typename T> визначає шаблонний параметр T
- myMax(T a, T b) працює з будь-яким типом даних, який підтримує оператор >
- Під час виклику компілятор підставляє відповідний тип (int, double, char тощо)

```
7
7
g
```

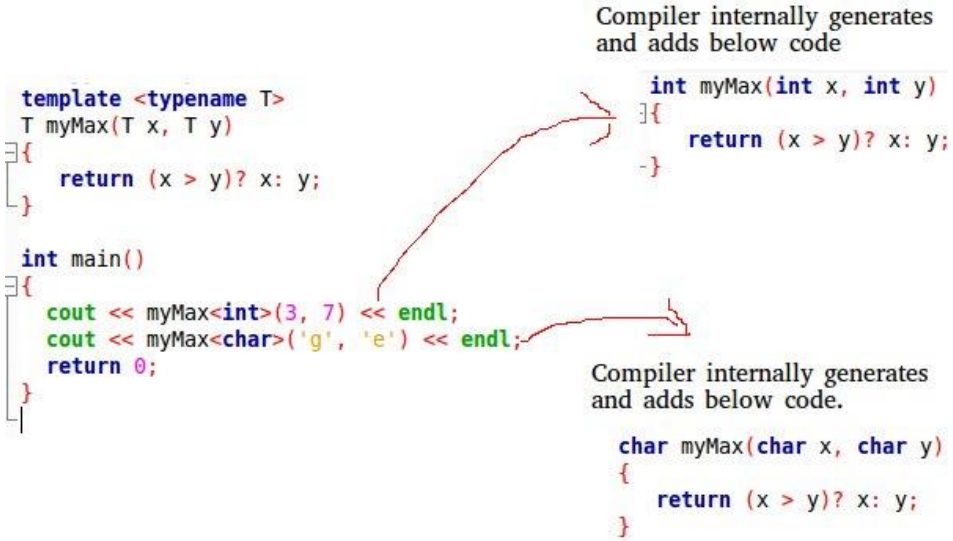
```
#include <iostream>
using namespace std;

// One function works for all data types. This
would work
// even for user defined types if operator '>'
is overloaded
template <typename T> T myMax(T x, T y)
{
    return (x > y) ? x : y;
}

int main()
{
    // Call myMax for int
    cout << myMax<int>(3, 7) << endl;
    // call myMax for double
    cout << myMax<double>(3.0, 7.0) << endl;
    // call myMax for char
    cout << myMax<char>('g', 'e') << endl;

    return 0;
}
```

Розгортання під час компіляції

Шаблони розгортаються **під час компіляції**, подібно до макросів. Однак різниця полягає в тому, що компілятор виконує перевірку типів **перед** розгортанням шаблону.

Ідея проста: у вихідному коді міститься лише одна функція або клас, але **після компіляції** код може містити **кілька копій цієї функції** або класу **для різних типів даних**.

Template instantiation (інстанціювання шаблонів) — це процес, під час якого компілятор створює конкретну версію шаблонної функції або класу на основі переданих аргументів типу.



Виведення типу аргументів шаблонів

- **Дедукція аргументів (template arguments deduction)** – виведення аргументів шаблону при відсутності явного їх задання:
  - на основі аргументів виклику функції
  - з контексту використання
- При дедукції **неявне перетворення типу аргументів виклику не відбувається**
  ```
  cout << myMax(3.2, 7) << endl;  // Comp. error: 'myMax': no matching overloaded function found
  ```
- У **явно спеціалізованій** шаблонній функції **допустиме неявне перетворення типу аргументів виклику**
  ```
  cout << myMax<double>(3.2, 7) << endl; // OK
  ```

Неявне інстанціювання

- **Implicit Instantiation (Неявне інстанціювання) -** Цей підхід означає, що компілятор автоматично створює необхідну версію шаблону під час його використання в коді. Інстанціювання відбувається тільки тоді, коли шаблон реально використовується

```cpp
template <typename T>
void printValue(T value) {
    std::cout << "Value: " << value << std::endl;
}

int main() {
    printValue(10);    // Компілятор створить printValue<int>
    printValue(3.14);  // Компілятор створить printValue<double>

    return 0;
}
```

Явне інстанціювання

- **Explicit Instantiation (Явне інстанціювання) -** При явному інстанціюванні ми примусово вказуємо компілятору створити певну версію шаблону, навіть якщо вона не використовується в коді. Використовується **для оптимізації компіляції**, коли потрібно створити конкретні інстанції шаблону передчасно

```
// Явне інстанціювання для int та double
template void printValue<int>(int);
template void printValue<double>(double);
```

Спеціалізація шаблонної функції

- **Спеціалізація** дозволяє створити унікальну версію шаблонної функції для певного типу, яка відрізняється від загального варіанту. Це використовується, коли певний тип вимагає особливої реалізації.

```cpp
template <typename T>
void printValue(T value) {
    std::cout << "General template: " << value << std::endl;
}

// Спеціалізація для типу int
template <>
void printValue<int>(int value) {
    std::cout << "Specialized template for int: " << value << std::endl;
}

int main() {
    printValue(10);      // Викличе спеціалізовану функцію
    printValue(3.14);    // Викличе загальний шаблон
    printValue("Hello"); // Викличе загальний шаблон

    return 0;
}
```

Передавання декількох шаблонних параметрів

Як і звичайні параметри, до шаблонів можна передавати більше ніж один тип даних як аргументи.

```cpp
template <typename T1, typename T2>
void logMessage(T1 level, T2 message) {
    std::cout << "[" << level << "] " << message << std::endl;
}

int main() {
    logMessage("INFO", "Application started.");
    logMessage("ERROR", 404);
    logMessage("DEBUG", 3.14);

    return 0;
}
```

Параметри за замовчуванням

Можна задавати значення параметрів типу за замовчуванням для кожного параметра шаблону:

```
template <typename T1, typename T2 = double>
void print(T1 a, T2 b = T2()) {
    cout << "First parameter: " << a << ", Second parameter: " << b << endl;
}

int main() {
    print(42);           // Викликається з параметром за замовчуванням для другого аргументу
    print(5, 10);        // Викликається з явно вказаними типами для обох параметрів
    print(3.14, 2.71);   // Викликається з явно вказаними типами для обох параметрів

    return 0;
}
```

Перевантаження шаблонних функцій

Можна створити кілька шаблонних функцій з різними параметрами:

```cpp
template <typename T>
void printValue(T value) {
    std::cout << "General template: " << value << std::endl;
}

template <typename T, typename U>
void printValue(T value1, U value2) {
    std::cout << "Overloaded template: " << value1 << " and " << value2 << std::endl;
}

int main() {
    printValue(10);          // Викличе загальний шаблон
    printValue(3.14, "Pi"); // Викличе перевантажений шаблон

    return 0;
}
```

Перевантаження шаблонної та звичайної функцій

Можна перевантажити шаблонну функцію звичайною функцією для певного типу. Звичайна функція має вищий пріоритет над шаблонною.

```
template <typename T>
void printValue(T value) {
    std::cout << "General template: " << value << std::endl;
}

// Звичайна функція для int
void printValue(int value) {
    std::cout << "Non-template function for int: " << value << std::endl;
}

int main() {
    printValue(10);       // Викличе звичайну функцію
    printValue(3.14);     // Викличе шаблонну функцію
    printValue("Hello"); // Викличе шаблонну функцію

    return 0;
}
```

Умовна спеціалізація шаблонної функції

Умовна спеціалізація — це спосіб вибору конкретної реалізації шаблонного класу або функції залежно від типу шаблонного аргументу. Для реалізації можна використовувати **std::enable_if** - **SFINAE (Substitution Failure Is Not An Error)**.

```cpp
template <typename T1, typename T2>
void func(T1 a, T2 b) {
    std::cout << "General template: " << a << " and " << b << std::endl;
}

// Спеціалізована функція, яка викликається тільки якщо обидва типи однакові
template <typename T, typename std::enable_if<std::is_same<T, T>::value, int>::type = 0>
void func(T a, T b) {
    std::cout << "Specialized template (same types): " << a << " and " << b << std::endl;
}

int main() {
    func(10, 3.14);   // Викличе загальний шаблон
    func(5, 7);       // Викличе спеціалізовану функцію
    func("Hello", "World"); // Викличе спеціалізовану функцію

    return 0;
}
```

Приклад шаблонної функції сортування

```cpp
template <typename T>
void displayArray(T* arr, int size) {
    for (int i = 0; i < size; ++i) {
        std::cout << arr[i] << std::endl;
    }
}

template <typename T>
void bubbleSort(T* arr, int size) {
    for (int i = 0; i < size - 1; ++i) {
        bool swapped = false;
        for (int j = 0; j < size - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                T temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped) {
            break;
        }
    }
}
```

```cpp
int main() {
    // Dynamic array of complex numbers
    int complexSize = 4;
    Complex* complexArr = new Complex[complexSize]{
        Complex(3, 4), Complex(1, 2), Complex(4, 5), Complex(2, 1)
    };

    std::cout << "\nComplex Numbers Before Sorting:\n";
    displayArray(complexArr, complexSize);

    bubbleSort(complexArr, complexSize);

    std::cout << "\nSorted Complex Numbers (by modulus):\n";
    displayArray(complexArr, complexSize);

    delete[] complexArr; // Don't forget to free the memory

    // Dynamic array of real numbers
    int realSize = 5;
    double* realArr = new double[realSize] {5.6, 2.3, 9.1, 3.4, 7.2};

    std::cout << "\nReal Numbers Before Sorting:\n";
    displayArray(realArr, realSize);

    // Sort real numbers
    bubbleSort(realArr, realSize);

    std::cout << "\nSorted Real Numbers:\n";
    // Display the sorted real numbers using template function
    displayArray(realArr, realSize);

    delete[] realArr; // Don't forget to free the memory

    return 0;
}
```

Шаблони класів

FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Шаблони класів

Шаблони класів, як і шаблони функцій, корисні, коли **клас визначає контейнер, що не залежить від типу вмістимих даних**. Вони можуть бути корисними для таких класів, як LinkedList, BinaryTree, Stack, Queue, Array тощо.

І перевантаження функцій, і **шаблони** є **прикладами статичного поліморфізму в ООП**. Перевантаження функцій використовується, коли кілька функцій виконують схожі (але не ідентичні) операції, тоді як **шаблони** застосовуються, коли **кілька функцій виконують ідентичні операції для різних типів даних**

Шаблон класу динамічного масиву

```cpp
template <typename T>
class DynamicArray {
private:
    T* arr;
    size_t size;
public:
    DynamicArray(size_t n) : size(n) {
        arr = new T[size];
    }
    ~DynamicArray() {
        delete[] arr;
    }
    T& operator[](size_t index) {
        return arr[index];
    }
    friend ostream& operator<<(ostream& os, const DynamicArray<T>& dynamicArray) {
        for (size_t i = 0; i < dynamicArray.size; ++i) {
            os << dynamicArray.arr[i] << " ";
        }
        return os;
    }
    T sum() const;
};

template <typename T>
T DynamicArray<T>::sum() const {
    T total = 0;
    for (size_t i = 0; i < size; ++i) {
        total += arr[i];
    }
    return total;
}
```

```cpp
int main() {
    DynamicArray<int> arr1(5);
    for (int i = 0; i < 5; ++i) {
        arr1[i] = (i + 1) * 10;
    }

    DynamicArray<double> arr2(3);
    arr2[0] = 1.1;
    arr2[1] = 2.2;
    arr2[2] = 3.3;

    cout << "Array 1: " << arr1 << endl;
    cout << "Array 1 Sum = " << arr1.sum() << endl;
    cout << "Array 2: " << arr2 << endl;
    cout << "Array 2 Sum = " << arr2.sum() << endl;

    return 0;
}
```

Instantiation - утворення екземпляру шаблону нового типу DynamicArray<int> та об’єкта цього типу:

DynamicArray<int> arr1(5);

Параметр non-type в шаблоні

```
template <class T, size_t size> // size є параметром non-type в шаблоні класу
class StaticArray
{
private:
    // Параметр non-type в шаблоні класу відповідає за розмір виділеного масиву
    T m_array[size];

public:
    T sum() const;

    T& operator[](int index)
    {
        return m_array[index];
    }
    friend ostream& operator<<(ostream& os, const StaticArray<T, size>& staticArray) {
        for (size_t i = 0; i < size; ++i) {
            os << staticArray.m_array[i] << " ";
        }
        return os;
    }
};

template <typename T, size_t size>
T StaticArray<T, size>::sum() const {
    T total = 0;
    for (size_t i = 0; i < size; ++i) {
        total += m_array[i];
    }
    return total;
}
```

```
// Оголошуємо цілочисельний масив з 10 елементів
StaticArray<int, 10> intArray;
```

Повна спеціалізація шаблону класу

```
// Спеціалізація для типу char
template <>
class DynamicArray<char> {
private:
    char* arr;
    size_t size;

public:
    DynamicArray(size_t n) : size(n) {
        arr = new char[size];
    }
    ~DynamicArray() {
        delete[] arr;
    }
    char& operator[](size_t index) {
        return arr[index];
    }

    // Перевантажений оператор виведення для типу char (виведення як рядок)
    friend ostream& operator<<(ostream& os, const DynamicArray& array) {
        for (size_t i = 0; i < array.size; ++i) {
            os << array.arr[i];  // Виведення символів без пробілів
        }
        return os;
    }
};
```

```
Array of integers: 0 2 4 6 8
Array of chars: Hello
```

Часткова спеціалізація шаблону класу

```cpp
// Часткова спеціалізація для вказівників (T*)
template <typename T>
class DynamicArray<T*> {
private:
    T** arr;
    size_t size;

public:
    // Конструктор
    DynamicArray(size_t n) : size(n) {
        arr = new T * [size];
        for (size_t i = 0; i < size; ++i) {
            arr[i] = new T();  // Виділення пам'яті для кожного елемента
        }
    }

    ~DynamicArray() {
        for (size_t i = 0; i < size; ++i) {
            delete arr[i];  // Видалення кожного об'єкта
        }
        delete[] arr;
    }

    T*& operator[](size_t index) {
        return arr[index];
    }

    friend ostream& operator<<(ostream& os, const DynamicArray& array) {
        for (size_t i = 0; i < array.size; ++i) {
            os << *(array.arr[i]) << " ";  // Вивід значень, на які вказують вказівники
        }
        return os;
    }
};
```

Дякую

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY