Шаблони функцій

Хоча функції і класи є потужними і гнучкими інструментами для ефективного програмування, в деяких випадках вони обмежені через вимоги мови C++ вказувати типи всіх використовуваних параметрів. Наприклад, припустимо, що нам потрібно написати функцію для визначення найбільшого числа серед двох чисел:

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

Все чудово до тих пір, поки ми працюємо з цілочисельними значеннями. А якщо нам доведеться працювати і зі значеннями типу double? Ви, ймовірно, вирішите перевантажити функцію max() для роботи з типом double:

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

Тепер у нас є дві версії однієї функції, які працюють з типами char, int, double і, якщо ми перевантажимо оператор >, навіть з класами! Однак, оскільки мова C++ вимагає, щоб ми вказували типи наших змінних, нам доводиться записувати кілька версій однієї і тієї ж функції, де єдине, що змінюється — це тип параметрів.

А це, в свою чергу, головний біль для програмістів, тому що підтримувати такий код непросто. І найважливіше те, що це порушує одну з концепцій ефективного програмування — зменшити до мінімуму дублювання коду.

У мові C++ **шаблони функцій — це функції, які служать взірцем для створення інших подібних функцій**. Головна ідея — створення функцій без вказівки точного типу(ів) деяких або всіх змінних. Для цього ми визначаємо функцію, вказуючи **тип параметра шаблону**, який використовується замість будь-якого типу даних. Після того, як ми створили функцію з типом параметра шаблону, ми фактично створили «трафарет функції».

При виклику шаблону функції, компілятор використовує «трафарет» в якості зразка функції, замінюючи тип параметра шаблону на фактичний тип змінних, переданих у функцію! Таким чином, ми можемо створити 50 «відтінків» функції, використовуючи всього лише один шаблон!

Створення шаблонів функцій

Розглянемо ще раз цілочисельну версію функції max():

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

Тут ми тричі вказуємо тип даних: в параметрах a, b і в типі повернення функції. Для створення шаблону цієї функції нам потрібно замінити тип int на тип параметра шаблону функції. Оскільки в цьому випадку використовується тільки один тип даних (int), то нам потрібно вказати тільки один тип параметра шаблону.

Ми можемо назвати цей тип як завгодно, головне, щоб це не було **зарезервованим/ключовим словом**. У мові C++ прийнято називати типи параметрів шаблонів великою літерою T (скор. від *"**T**ype"*).

Ось наша перероблена функція max():

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

Але це ще не все. Програма не працюватиме, тому що компілятор не знає, що таке T!

Щоб все запрацювало, нам потрібно повідомити компілятору дві речі:

Визначення шаблону функції.

Вказівка того, що T є типом параметра шаблону функції.

Ми можемо зробити це в одному рядку коду, виконавши оголошення шаблону (а точніше — оголошення параметрів шаблону):

```
template <typename T> // оголошення параметра шаблону функції
T max(T a, T b)
{
    return (a > b) ? a : b;
}
```

Розглянемо детально оголошення параметрів шаблону:

Спочатку пишемо ключове слово template, яке повідомляє компілятору, що далі ми будемо оголошувати параметри шаблону.

Параметри шаблону функції вказуються в кутових дужках (<>).

Для створення типів параметрів шаблону використовуються ключові слова typename і class. В базових випадках використання шаблонів функцій різниці між typename і class немає, тому ви можете вибрати будь-яке з двох. Якщо ви використовуєте ключове слово class, то фактичний тип параметрів не обов'язково повинен бути класом (це може бути змінна фундаментального типу даних, вказівник або щось інше).

Потім даємо назву типу параметра шаблону (зазвичай T).

Якщо потрібно кілька типів параметрів шаблону, то вони розділяються комами:

```
template <typename T1, typename T2>
// Шаблон функції тут
```

Якщо параметрів кілька, то їх зазвичай називають T1, T2 або іншими літерами: T, S.

Примітка: Оскільки тип аргументу функції, що передається в тип T, може бути класом, а класи, як правило, не рекомендується передавати по значенню, то краще зробити параметри і значення, що повертається, нашого шаблону функції константними посиланнями, наприклад:

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

Використання шаблонів функцій

Використання шаблонів функцій аналогічне використанню звичайних функцій:

```
#include <iostream>

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

int main()
{
    int i = max(4, 8);
    std::cout << i << '\n';

    double d = max(7.56, 21.434);
    std::cout << d << '\n';

    char ch = max('b', '9');
    std::cout << ch << '\n';

    return 0;
}
```

Результат:

```
8
21.434
b
```

Зверніть увагу, всі три виклики функції max() мають параметри різних типів! Оскільки ми викликаємо функцію max() з трьома різними типами параметрів, то компілятор використовує шаблон функції для створення трьох різних версій функції max():

Версія з параметрами типу int (max<int>).

Версія з параметрами типу double (max<double>).

Версія з параметрами типу char (max<char>).

Нам не потрібно явно вказувати тип переданих значень (частина <int> в max<int>), компілятор визначить це самостійно.

Висновки

Шаблони функцій економлять багато часу, тому що шаблон ми пишемо тільки один раз, а використовувати можемо з різними типами даних. Як тільки ви звикнете до написання шаблонів функцій, ви виявите, що по часу це займає не більше написання звичайної функції (однієї версії звичайної функції). Шаблони функцій набагато спрощують подальшу підтримку коду, і вони безпечніші, тому що немає необхідності виконувати вручну перевантаження функції, копіюючи код і змінюючи лише типи даних, коли потрібна підтримка нового типу даних.

У шаблонів функцій є кілька недоліків, і було б неприпустимо, якби ми про них не поговорили:

По-перше, деякі старі компілятори можуть не підтримувати шаблони функцій або підтримувати, але з обмеженнями. Однак зараз це вже не така проблема, як раніше.

По-друге, шаблони функцій часто видають божевільні повідомлення про помилки, які набагато складніше розшифрувати, ніж помилки звичайних функцій.

По-третє, шаблони функцій можуть збільшити час компіляції і розмір коду, тому що один шаблон може бути «реалізований» і перекомпільований в декількох файлах.

Дані недоліки досить незначні в порівнянні з потужністю і гнучкістю шаблонів функцій!

Мова C++ не компілює шаблони функцій напряму. Замість цього, коли компілятор зустрічає виклик шаблону функції, він копіює шаблон функції і замінює типи параметрів шаблону функції фактичними (переданими) типами даних. Функція з фактичними типами даних називається екземпляром шаблону функції (або «об'єктом шаблону функції»).

Розглянемо це на практиці. По-перше, створимо шаблон функції:

```
template <typename T> // оголошення параметра шаблону функції
const T& max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

Потім зробимо виклик шаблону функції:

```
int i = max(4, 8); // викликається max(int, int)
```

Компілятор бачить, що обидва числа є цілочисельними, тому він копіює шаблон функції і створює екземпляр шаблону max(int, int):

```
    const int& max(const int& a, const int& b)
{
    return (a > b) ? a : b;
}
```

Це тепер уже «звичайна функція». Припустимо, що нам потрібно знову викликати функцію max(), але вже з іншим типом даних:

```
double d = max(7.58, 19.378); // викликається max(double, double)
```

Мова C++ автоматично створює екземпляр шаблону max(double, double):

```
  const double& max(const double& a, const double& b)
{
    return (a > b) ? a : b;
}
```

І потім компілює його. Також варто відзначити, що, якщо ви створите шаблон функції, але не викличете його, екземпляри цього шаблону не будуть створені.

Оператори, виклики функцій і шаблони функцій

Шаблони функцій працюють як з фундаментальними типами даних (char, int, double тощо), так і з класами (але є нюанс). Екземпляр шаблону компілюється як звичайна функція. У звичайній функції будь-які оператори або виклики інших функцій, які використовуються в цій функції, повинні бути визначені/перевантажені, або ви отримаєте помилку компіляції. Аналогічно, будь-які оператори або виклики інших функцій, які присутні в шаблоні функції, повинні бути визначені/перевантажені для роботи з фактичними (переданими) типами даних. Розглянемо це на практиці.

По-перше, створимо простий клас:

```
class Dollars
{
private:
    int m_dollars;
public:
    Dollars(int dollars)
        : m_dollars(dollars)
    {
```

```
    }
};
```

Тепер подивимося, що станеться при спробі виклику функції max() з об'єктами класу Dollars:

```
template <typename T> // оголошення параметра шаблону функції
const T& max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}

class Dollars
{
private:
    int m_dollars;
public:
    Dollars(int dollars)
        : m_dollars(dollars)
    {
    }
};

int main()
{
    Dollars seven(7);
    Dollars twelve(12);

    Dollars bigger = max(seven, twelve);

    return 0;
}
```

Мова C++ створить наступний екземпляр шаблону функції max():

```
const Dollars& max(const Dollars& a, const Dollars& b)
{
    return (a > b) ? a : b;
}
```

А потім компілятор спробує скомпілювати цю функцію, але нічого не вийде, тому що C++ не знає, як обробляти вираз a > b! Отже, це призведе до помилки:

Помилка C2676 бінарний ">": "const T" не визначає цей оператор або перетворення до типу припустимо до вбудованого оператора

Повідомлення про помилку вказує на той факт, що ми не перевантажили оператор > для класу Dollars. Давайте перевантажимо:

```
class Dollars
{
private:
    int m_dollars;
public:
    Dollars(int dollars)
        : m_dollars(dollars)
    {
    }

    friend bool operator>(const Dollars& d1, const Dollars& d2)
    {
        return (d1.m_dollars > d2.m_dollars);
    }
};
```

Тепер мова C++ знає, як обробляти вираз a > b, коли в якості змінних використовуються об'єкти класу Dollars!

Ще один приклад

Створимо шаблон функції, яка обчислює середнє арифметичне елементів масиву:

```
template <class T>
T average(T* array, int length)
{
    T sum = 0;
    for (int count = 0; count < length; ++count)
        sum += array[count];

    sum /= length;
```

```
    return sum;
}
```

Протестуємо:

```
#include <iostream>

template <class T>
T average(T* array, int length)
{
    T sum = 0;
    for (int count = 0; count < length; ++count)
        sum += array[count];

    sum /= length;
    return sum;
}

int main()
{
    int array1[] = { 6, 4, 1, 3, 7 };
    std::cout << average(array1, 5) << '\n';

    double array2[] = { 4.25, 5.37, 8.44, 9.25 };
    std::cout << average(array2, 4) << '\n';

    return 0;
}
```

Результат:

```
4
6.8275
```

Як ви бачите, все відмінно працює з фундаментальними типами даних!

Оскільки тип повернення шаблону функції той же, що і тип переданих елементів масиву в функцію, то обчислення середнього арифметичного цілочисельних значень призведе до цілочисельного результату (з відкиданням будь-якої дробової частини), як і обчислення значень типу double призведе до результату типу double. Це може бути не очевидним, тому хорошим тоном буде вказати на це в коментарях.

Тепер подивимося, що станеться при виконанні функції average() з об’єктами класу Dollars:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars;
public:
    Dollars(int dollars)
        : m_dollars(dollars)
    {
    }

    friend bool operator>(const Dollars& d1, const Dollars& d2)
    {
        return (d1.m_dollars > d2.m_dollars);
    }
};

template <class T>
T average(T* array, int length)
{
    T sum = 0;
    for (int count = 0; count < length; ++count)
        sum += array[count];

    sum /= length;
    return sum;
}

int main()
{
```

```
    Dollars array3[] = { Dollars(7), Dollars(12), Dollars(18), Dollars(15) };
    std::cout << average(array3, 4) << '\n';

    return 0;
}
```

Результат:

чотири сторінки помилок

Компілятор збожеволів. Незважаючи на настільки об'ємний «результат», тут все досить просто. У перших рядках повідомляється, що компілятор не зміг знайти перевантаження оператора << для класу Dollars. Далі вказуються функції з типами даних, які викликалися для порівняння, але так і не підійшли. І в кінці вказуються параметр шаблону і фактичний тип параметра.

Visual Studio береже наші нерви і надає нам альтернативний вивід помилок:

Помилка E0349 відсутній оператор "<<", який відповідає цим операндам Помилка C2679 бінарний "<<": не знайдено оператор, який приймає правий операнд типу "T" (або прийнятне перетворення відсутнє)

Пам'ятайте, що average() повертає об'єкт класу Dollars, а ми намагаємося цей об'єкт вивести за допомогою оператора виводу << і std::cout. Однак, ми не перевантажили оператор << для класу Dollars. Давайте виправимо це:

```
class Dollars
            {
            private:
                int m_dollars;
            public:
                Dollars(int dollars)
                    : m_dollars(dollars)
                {
                }

                friend bool operator>(const Dollars& d1, const Dollars& d2)
                {
                    return (d1.m_dollars > d2.m_dollars);
                }

                friend std::ostream& operator<< (std::ostream& out, const Dollars& dollars)
                {
                    out << dollars.m_dollars << " dollars ";
                    return out;
                }
            };
```

Якщо ж тепер запустити програму, то отримаємо наступне:

Помилка C2676 бінарний "+=": "T" не визначає цей оператор або перетворення до типу припустимо до вбудованого оператора Помилка C2676 бінарний "/=": "T" не визначає цей оператор або перетворення до типу припустимо до вбудованого оператора

Ці помилки були викликані екземпляром шаблону функції, створеним при виклику average(Dollars*, int). Пам'ятайте, що при виклику шаблону функції, компілятор копіює шаблон функції з типами параметрів, а потім замінює типи параметрів шаблону на фактичні (передані) типи даних. Ось екземпляр шаблону функції average(), де T є класом Dollars:

```
template <class Dollars>
            Dollars average(Dollars* array, int length)
            {
                Dollars sum = 0;
                for (int count = 0; count < length; ++count)
                    sum += array[count];

                sum /= length;
                return sum;
            }
```

Причина, по якій ми отримали повідомлення про помилку, криється в наступному рядку:

```
sum += array[count];
```

В цьому випадку sum є об'єктом класу Dollars. А щоб все запрацювало, нам потрібно перевантажити оператори += і /= для класу Dollars:

```
class Dollars
            {
            private:
                int m_dollars;
```

```
public:
    Dollars(int dollars)
        : m_dollars(dollars)
    {
    }

    friend bool operator>(const Dollars& d1, const Dollars& d2)
    {
        return (d1.m_dollars > d2.m_dollars);
    }

    friend std::ostream& operator<< (std::ostream& out, const Dollars& dollars)
    {
        out << dollars.m_dollars << " dollars ";
        return out;
    }

    Dollars& operator+=(Dollars dollars)
    {
        m_dollars += dollars.m_dollars;
        return *this;
    }

    Dollars& operator/=(int value)
    {
        m_dollars /= value;
        return *this;
    }
};
```

Нарешті, наш код скомпілюється, і результат:

```
13 dollars
```

Хоча виконана робота може здатися дуже великою, але це тільки через те, що наш клас Dollars з самого початку був “худий як тріска”. Ключовий момент заключається в тому, що нам не потрібно модифікувати average(), щоб він працював з об’єктами класу Dollars (або з будь-яким іншим типом даних). Вся робота була пророблена тільки з класом Dollars, а про все інше компілятор подбав самостійно!

Шаблони і контейнерні класи

Розглянемо клас ArrayInt:

```
#ifndef ARRAYINT_H
#define ARRAYINT_H

#include <assert.h> // для assert()

class ArrayInt
{
private:
    int m_length;
    int* m_data;

public:
    ArrayInt()
    {
        m_length = 0;
        m_data = nullptr;
    }

    ArrayInt(int length)
    {
        assert(length > 0);
        m_data = new int[length];
        m_length = length;
    }

    ~ArrayInt()
    {
        delete[] m_data;
    }
```

```
    void Erase()
    {
        delete[] m_data;
        // Присвоюємо значення nullptr для m_data, щоб на виході не отримати висячий вказівник!
        m_data = nullptr;
        m_length = 0;
    }

    int& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() { return m_length; }
};

#endif
```

Хоча цей клас забезпечує простий спосіб створення масиву цілочисельних значень, але що, якщо нам потрібно буде працювати зі значеннями типу double? Використовуючи традиційні методи програмування ми створили б новий клас ArrayDouble для роботи зі значеннями типу double:

```
#ifndef ARRAYDOUBLE_H
#define ARRAYDOUBLE_H

#include <assert.h> // для assert()

    class ArrayDouble
{
private:
    int m_length;
    double* m_data;

public:
    ArrayDouble()
    {
        m_length = 0;
        m_data = nullptr;
    }

    ArrayDouble(int length)
    {
        assert(length > 0);
        m_data = new double[length];
        m_length = length;
    }

    ~ArrayDouble()
    {
        delete[] m_data;
    }

    void Erase()
    {
        delete[] m_data;
        // Присвоюємо значення nullptr для m_data, щоб на виході не отримати висячий вказівник!
        m_data = nullptr;
        m_length = 0;
    }

    double& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() { return m_length; }
};

#endif
```

Хоча коду багато, але класи майже ідентичні, змінюється тільки тип даних! Як ви вже могли б здогадатися, це ідеальний випадок для використання шаблонів. Створення шаблону класу аналогічно створенню шаблону функції. Наприклад, створимо шаблон класу Array:

Array.h:

```
#ifndef ARRAY_H
#define ARRAY_H

#include <assert.h> // для assert()

    template <class T> // це шаблон класу з T замість фактичного (переданого) типу даних
class Array
{
private:
    int m_length;
    T* m_data;

public:
    Array()
    {
        m_length = 0;
        m_data = nullptr;
    }

    Array(int length)
    {
        m_data = new T[length];
        m_length = length;
    }

    ~Array()
    {
        delete[] m_data;
    }

    void Erase()
    {
        delete[] m_data;
        // Присвоюємо значення nullptr для m_data, щоб на виході не отримати висячий вказівник!
        m_data = nullptr;
        m_length = 0;
    }


    T& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    // Довжина масиву завжди є цілочисельним значенням, вона не залежить від типу елементів
масиву
    int getLength(); // визначаємо метод і шаблон методу getLength() нижче
};

template <typename T> // метод, визначений поза тілом класу, потребує власного визначення
шаблону методу
int Array<T>::getLength() { return m_length; } // зверніть увагу, ім'я класу - Array<T>, а не
просто Array

#endif
```

Як ви можете бачити, ця версія майже ідентична версії ArrayInt, за винятком того, що ми додали оголошення параметра шаблону класу і змінили тип даних з int на T.

Зверніть увагу, ми визначили функцію getLength() поза тілом класу. Це необов’язково, але початківці зазвичай спотикаються на цьому через синтаксис. Кожен метод шаблону класу, оголошений поза тілом класу, потребує власного оголошення шаблону. Також зверніть увагу, що ім’я шаблону класу — Array<T>, а не Array (Array вказуватиме на НЕ шаблонну версію класу Array).

Ось приклад використання шаблону класу Array:

```
#include <iostream>
```

```
#include "Array.h"

int main()
{
    Array<int> intArray(10);
    Array<double> doubleArray(10);

    for (int count = 0; count < intArray.getLength(); ++count)
    {
        intArray[count] = count;
        doubleArray[count] = count + 0.5;
    }

    for (int count = intArray.getLength() - 1; count >= 0; --count)
        std::cout << intArray[count] << "\t" << doubleArray[count] << '\n';

    return 0;
}
```

Результат:

```
9       9.5
8       8.5
7       7.5
6       6.5
5       5.5
4       4.5
3       3.5
2       2.5
1       1.5
0       0.5
```

Шаблони класів працюють точно так же, як і шаблони функцій: компілятор копіює шаблон класу, замінюючи типи параметрів шаблону класу на фактичні (передані) типи даних, а потім компілює цю копію. Якщо у вас є шаблон класу, але ви його не використовуєте, то компілятор не буде його навіть компілювати.

Шаблони класів ідеально підходять для реалізації контейнерних класів, тому що дуже часто таким класам доводиться працювати з різними типами даних, а шаблони дозволяють це організувати в мінімальній кількості коду. Хоча синтаксис трохи потворний, і повідомлення про помилки іноді можуть бути «об'ємними», шаблони класів дійсно є однією з кращих і найбільш корисних конструкцій мови C++.

Шаблони класів в Стандартній бібліотеці C++

Тепер ви вже зрозуміли, чим насправді є std::vector<int>? Правильно, std::vector — це шаблон класу, а int — це всього лише переданий тип даних! Стандартна бібліотека C++ містить багато визначених шаблонів класів, доступних для вашого використання.

Шаблони класів і Заголовкові файли

Шаблон не є ані класом, ані функцією — це трафарет, який використовується для створення класів або функцій. Таким чином, шаблони працюють не так, як звичайні функції або класи. У більшості випадків це не є проблемою, але на практиці трапляються різні ситуації.

Працюючи зі звичайними класами ми поміщаємо визначення класу в заголовковий файл, а визначення методів цього класу в окремий .cpp-файл з аналогічним ім'ям. Таким чином, фактичне визначення класу компілюється як окремий файл всередині проекту. Однак з шаблонами все відбувається дещо інакше.

Розглянемо наступне:

Array.h:

```
#ifndef ARRAY_H
#define ARRAY_H

#include <assert.h> // для assert()

    template <class T>
class Array
```

```
{
private:
    int m_length;
    T* m_data;

public:
    Array()
    {
        m_length = 0;
        m_data = nullptr;
    }

    Array(int length)
    {
        m_data = new T[length];
        m_length = length;
    }

    ~Array()
    {
        delete[] m_data;
    }

    void Erase()
    {
        delete[] m_data;
        // Присвоюємо значення nullptr для m_data, щоб на виході не отримати висячий вказівник!
        m_data = nullptr;
        m_length = 0;
    }


    T& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    // Довжина масиву завжди є цілочисельним значенням, вона не залежить від типу елементів
масиву
    int getLength();
};

#endif
```

Array.cpp:

```
#include "Array.h"

template <typename T>
int Array<T>::getLength() { return m_length; }
```

main.cpp:

```
#include "Array.h"

int main()
{
    Array<int> intArray(10);
    Array<double> doubleArray(10);

    for (int count = 0; count < intArray.getLength(); ++count)
    {
        intArray[count] = count;
        doubleArray[count] = count + 0.5;
    }

    for (int count = intArray.getLength() - 1; count >= 0; --count)
        std::cout << intArray[count] << "\t" << doubleArray[count] << '\n';

    return 0;
}
```

Вищенаведена програма скомпілюється, але викличе наступну помилку лінкера:

unresolved external symbol "public: int __thiscall Array::getLength(void)" (?GetLength@?$Array@H@@QAEHXZ)

Чому так? Зараз розберемося.

Для використання шаблону компілятор повинен бачити як визначення шаблону (а не тільки оголошення), так і тип шаблону, який застосовується для створення екземпляру шаблону. Пам'ятаємо, що мова C++ компілює файли окремо. Коли заголовок Array.h підключається в main.cpp, то визначення шаблону класу копіюється в цей файл. У main.cpp компілятор бачить, що нам потрібні два екземпляри шаблону класу: Array<int> і Array<double>, він створить їх, а потім скомпілює весь цей код як частину файлу main.cpp. Однак, коли справа дійде до компіляції Array.cpp (окремим файлом), компілятор забуде, що ми використовували Array<int> і Array<double> в main.cpp і не створить екземпляр шаблону функції getLength(), який нам потрібен для виконання програми. Ми отримаємо помилку лінкера, тому що компілятор не зможе знайти визначення Array<int>::getLength() або Array<double>::getLength().

Цю проблему можна вирішити кількома способами.

Найпростіший варіант — помістити код з Array.cpp в Array.h нижче класу. Таким чином, коли ми підключатимемо Array.h, весь код шаблону класу (повне оголошення і визначення як класу, так і його методів) знаходитиметься в одному місці. Плюс цього способу — простота. Мінус — якщо шаблон класу використовується в багатьох місцях, то ми отримаємо багато локальних копій шаблону класу, що збільшить час компіляції і лінкінгу файлів (лінкер повинен буде видалити дублювання визначень класу і методів, щоб виконуваний файл не був «занадто роздутим»). Рекомендується використовувати це рішення до тих пір, поки час компіляції або лінкінгу не є проблемою.

Якщо ви вважаєте, що розміщення коду з Array.cpp в Array.h зробить Array.h занадто великим/безладним, то альтернативою буде перейменування Array.cpp в Array.inl (.inl від англ. "inline" = "вбудований"), а потім підключення Array.inl з нижньої частини файлу Array.h. Це дасть той же результат, що і розміщення всього коду в заголовку, але таким чином код вийде трохи чистішим.

Є ще одне рішення — підключення .cpp-файлів, але цей варіант не рекомендується використовувати через нестандартне застосування директиви #include.

Ще один альтернативний варіант — використовувати підхід трьох файлів:

Визначення шаблону класу зберігається в заголовковому файлі.

Визначення методів шаблону класу зберігаються в окремому .cpp-файлі.

Потім додаємо третій файл, який містить всі необхідні нам екземпляри шаблону класу.

Наприклад, templates.cpp:

```
// Таким чином, ми гарантуємо, що компілятор побачить повне визначення шаблону класу Array
#include "Array.h"
#include "Array.cpp" // ми порушуємо правила хорошого тону в програмуванні, але тільки в цьому місці

// Тут ви #include інші файли .h і .cpp з визначеннями шаблонів, які вам потрібні

template class Array<int>; // явно створюємо екземпляр шаблону класу Array<int>
template class Array<double>; // явно створюємо екземпляр шаблону класу Array<double>

// Тут ви явно створюєте інші екземпляри шаблонів, які вам потрібні
```

Частина template class змусить компілятор явно створити зазначені екземпляри шаблону класу. У прикладі, наведеному вище, компілятор створить Array<int> і Array<double> всередині templates.cpp. Оскільки templates.cpp знаходиться всередині нашого проекту, то він скомпілюється і вдало зв'яжеться з іншими файлами (відбудеться лінкінг).

Цей метод ефективніший, але вимагає створення/підтримки третього файлу (templates.cpp) для кожної з ваших програм (проектів) окремо.

Параметр non-type

Параметр non-type в шаблоні — це спеціальний параметр шаблону, який замінюється не типом даних, а конкретним значенням. Цим значенням може бути:

цілочисельне значення або перерахування;

вказівник або посилання на об'єкт класу;

вказівник або посилання на функцію;

вказівник або посилання на метод класу;

std::nullptr_t.

У наступному прикладі ми створимо шаблон класу StaticArray, який використовує як параметр типу, так і параметр non-type. Параметр типу відповідає за тип даних елементів статичного масиву, а параметр non-type відповідає за розмір виділеного масиву:

```
#include <iostream>
```

```cpp
template <class T, int 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* getArray();

    T& operator[](int index)
    {
        return m_array[index];
    }
};

// Синтаксис визначення шаблону методу і самого методу поза тілом класу з параметром non-type
template <class T, int size>
T* StaticArray<T, size>::getArray()
{
    return m_array;
}

int main()
{
    // Оголошуємо цілочисельний масив з 10 елементів
    StaticArray<int, 10> intArray;

    // Заповнюємо масив значеннями
    for (int count = 0; count < 10; ++count)
        intArray[count] = count;

    // Виводимо елементи масиву у зворотному порядку
    for (int count = 9; count >= 0; --count)
        std::cout << intArray[count] << " ";
    std::cout << '\n';

    // Оголошуємо масив типу double з 5 елементів
    StaticArray<double, 5> doubleArray;

    // Заповнюємо масив значеннями
    for (int count = 0; count < 5; ++count)
        doubleArray[count] = 5.5 + 0.1 * count;

    // Виводимо елементи масиву
    for (int count = 0; count < 5; ++count)
        std::cout << doubleArray[count] << ' ';

    return 0;
}
```

Результат виконання програми:

```
9 8 7 6 5 4 3 2 1 0
5.5 5.6 5.7 5.8 5.9
```

Нам навіть не довелося динамічно виділяти змінну-член m_array! Це пов’язано з тим, що для будь-якого створеного об’єкта класу StaticArray його розмір є конкретно заданим значенням (можна сказати константою), яке передає користувач. Наприклад, якщо ми створимо екземпляр StaticArray<int, 10>, то компілятор замінить змінну розміру масиву (size) на 10. Таким чином, ми отримаємо m_array типу int[10], який можна виділити статичним чином.

Цю особливість використовує вже відомий нам клас зі Стандартної бібліотеки C++ — std::array. Коли ми виділяємо std::array<int, 5>, то int є параметром типу, а 5 — параметром non-type в шаблоні класу!

При створенні екземпляра шаблону функції для певного типу даних компілятор копіює шаблон функції і замінює параметр типу шаблону функції на фактичний (переданий) тип даних. Це означає, що всі екземпляри функції мають одну реалізацію, але різні типи даних. Хоча в більшості випадків це саме

те, що потрібно, іноді може знадобитися, щоб реалізація шаблону функції для одного типу даних відрізнялася від реалізації шаблону функції для іншого типу даних.

Спеціалізація шаблонів саме для цього і призначена.

Розглянемо дуже простий шаблон класу:

```
template <class T>
class Repository
{
private:
    T m_value;
public:
    Repository(T value)
    {
        m_value = value;
    }

    ~Repository()
    {
    }

    void print()
    {
        std::cout << m_value << '\n';
    }
};
```

Вищенаведений код працює з багатьма типами даних:

```
int main()
{
    // Ініціалізуємо об'єкти класу
    Repository<int> nValue(7);
    Repository<double> dValue(8.4);

    // Виводимо значення об'єктів класу
    nValue.print();
    dValue.print();
}
```

Результат:

```
7
8.4
```

Тепер, припустимо, що нам потрібно, щоб значення типу double (тільки типу double) виводилися в експоненціальному записі. Для цього ми можемо використати спеціалізацію шаблону функції (або «повну/явну спеціалізацію шаблону функції») для створення окремої версії функції print() для виводу значень типу double.

Все просто: записуємо екземпляр шаблону функції (якщо функція є методом класу, то робимо це за межами класу), вказуючи потрібний нам тип даних. Наприклад, ось спеціальний шаблон функції print() для значень типу double:

```
template <>
void Repository<double>::print()
{
    std::cout << std::scientific << m_value << '\n';
}
```

Коли компілятору потрібно буде створити екземпляр Repository<double>::print(), він побачить, що ми вже явно визначили цю функцію, і тому він використовуватиме саме цей екземпляр, а не копіюватиме загальну для всіх типів даних версію шаблону функції print() .

Частина template <> повідомляє компілятору, що це шаблон функції, але без параметрів (тому що в цьому випадку ми явно вказуємо потрібний нам тип даних).

Результат виконання програми:

```
7
8.400000e+00
```

Ще один приклад

Розглянемо ще один випадок, де спеціалізація шаблонів функцій може бути корисна. Наприклад, що станеться, якщо ми спробуємо використати наш шаблон класу Repository з типом даних char*?

```
int main()
{
    // Динамічно виділяємо тимчасовий рядок
```

```
    char* string = new char[40];

    // Просимо користувача ввести своє ім'я
    std::cout << "Enter your name: ";
    std::cin >> string;

    // Зберігаємо те, що ввів користувач
    Repository<char*> repository(string);

    // Видаляємо тимчасовий рядок
    delete[] string;

    // Намагаємося вивести те, що ввів користувач
    repository.print(); // отримуємо сміття
}
```

Виявляється, замість виводу імені користувача, repository.print() виведе сміття! Чому?

При створенні екземпляра шаблону для типу char*, конструктор Repository<char*> виглядає наступним чином:

```
template <>
Repository<char*>::Repository(char* value)
{
    m_value = value;
}
```

Іншими словами, це просто присвоювання вказівника (поверхневе копіювання)! В результаті m_value вказує на ту ж область пам’яті, що і змінна string. А коли ми видаляємо string в main(), то ми видаляємо значення, на яке вказує і m_value! Таким чином, відбувається витік пам’яті, і ми отримуємо сміття при спробі виводу m_value.

На щастя, ми можемо це виправити, використовуючи явну спеціалізацію шаблону функції. Замість копіювання вказівника на string, нам потрібно, щоб конструктор копіював саме значення string. Напишемо окремий конструктор для типу даних char*, який саме це і робитиме:

```
template <>
Repository<char*>::Repository(char* value)
{
    // Визначаємо довжину value
    int length = 0;
    while (value[length] != '\0')
        ++length;
    ++length; // +1, враховуючи нуль-термінатор

    // Виділяємо пам'ять для зберігання значення value
    m_value = new char[length];

    // Копіюємо фактичне значення value в m_value
    for (int count = 0; count < length; ++count)
        m_value[count] = value[count];
}
```

Тепер при виділенні змінної типу Repository<char*> саме цей конструктор використовуватиметься замість стандартного. В результаті m_value отримає свою власну копію string. Отже, коли ми видалимо string в main(), m_value це ніяк не зачепить.

Однак, тепер клас має витік пам’яті для типу char*, оскільки m_value не буде видалений, коли змінна repository вийде з області видимості. Як ви вже могли здогадатися, це також можна вирішити, зробивши окремий деструктор для типу char*:

```
template <>
Repository<char*>::~Repository()
{
    delete[] m_value;
}
```

Тепер, коли змінні типу Repository<char*> вийдуть з області видимості, пам’ять, виділена в спеціальному конструкторі, буде видалена в спеціальному деструкторі.

Хоча у всіх прикладах, наведених вище, ми працюємо з методами класу, ви також можете аналогічно виконувати явну спеціалізацію шаблонів звичайних функцій (які не є методами класів).

Виявляється, ми можемо спеціалізувати не тільки шаблони функцій, але і шаблони класів.

Розглянемо клас-масив, який може зберігати 8 об’єктів:

```
template <class T>
class Repository8
```

```
{
private:
    T m_array[8];

public:
    void set(int index, const T& value)
    {
        m_array[index] = value;
    }

    const T& get(int index)
    {
        return m_array[index];
    }
};
```

Оскільки це шаблон класу, то він працюватиме з будь-яким типом даних:

```
#include <iostream>

int main()
{
    // Оголошуємо цілочисельний об'єкт-масив
    Repository8<int> intRepository;

    for (int count = 0; count < 8; ++count)
        intRepository.set(count, count);

    for (int count = 0; count < 8; ++count)
        std::cout << intRepository.get(count) << '\n';

    // Оголошуємо об'єкт-масив типу bool
    Repository8<bool> boolRepository;

    for (int count = 0; count < 8; ++count)
        boolRepository.set(count, count % 5);

    for (int count = 0; count < 8; ++count)
        std::cout << (boolRepository.get(count) ? "true" : "false") << '\n';

    return 0;
}
```

Результат:

```
0
1
2
3
4
5
6
7
false
true
true
true
true
false
true
true
```

Хоча все працює правильно, реалізація Repository8<bool>, насправді, не настільки ефективна, якою вона могла б бути. Оскільки всі змінні повинні мати адреси, а ЦП не може дати адресу чомусь меншому за 1 байт, то розмір всіх змінних повинен становити не менше 1 байту. Отже, кожна змінна типу bool займає цілий байт, хоча технічно їй потрібен тільки 1 біт для зберігання значення true або false! Таким чином, змінна типу bool — це 1 біт корисної інформації та 7 біт марно витраченого місця.

Виходить, що наш клас Repository8<bool>, який має 8 змінних типу bool, фактично працює тільки з 1 байтом даних, а решта 7 байтів витрачаються даремно.

Вихід є: ми можемо “стиснути” 8 змінних типу bool в 1 байт, заощадивши при цьому решту 7 байтів. Однак для цього нам потрібно буде змінити реалізацію класу, замінивши масив з 8 змінних типу bool (8 байтів) на 1 змінну типу unsigned char (1 байт). Ми могли б, звичайно, використати для цього новий окремий клас, але це неефективно, і програмісту доведеться пам’ятати, що Repository8<T> працює з усіма типами даних, крім bool, а при роботі з bool слід викликати Repository8Bool (неважливо, яке ім’я буде у цього класу). Це зайва робота, яку можна уникнути, використовуючи явну спеціалізацію шаблону класу.

Спеціалізація шаблону класу

Спеціалізація шаблону класу (або «явна спеціалізація шаблону класу») дозволяє спеціалізувати шаблон класу для роботи з певним типом даних (або відразу з декількома типами даних, якщо є кілька параметрів шаблону).

Спеціалізація шаблону класу розглядається компілятором як повністю окремий і незалежний клас, хоч і виділяється як звичайний шаблон класу. Це означає, що ми можемо змінити в класі все що завгодно, включаючи його реалізацію/методи/специфікатори доступу тощо.

Розглянемо спеціалізацію шаблону класу Repository8 для роботи з типом bool:

```
template <>
class Repository8<bool> // спеціалізуємо шаблон класу Repository8 для роботи з типом bool
{
    // Реалізація класу
private:
    unsigned char m_data;

public:
    Repository8() : m_data(0)
    {
    }

    void set(int index, bool value)
    {
        // Вибираємо оперований біт
        unsigned char mask = 1 << index;

        if (value) // якщо на вході у нас true, то біт потрібно "увімкнути"
            m_data |= mask; // використовуємо побітове АБО, щоб "увімкнути" біт
        else // якщо на вході у нас false, то біт потрібно "вимкнути"
            m_data &= ~mask; // використовуємо побітове І, щоб "вимкнути" біт
    }

    bool get(int index)
    {
        // Вибираємо біт
        unsigned char mask = 1 << index;

        // Використовуємо побітове І для отримання значення біта, а потім виконується його
неявна конвертація в тип bool
        return (m_data & mask) != 0;
    }
};
```

По-перше, починаємо з template<>. Ключове слово template повідомляє компілятору, що це шаблон, а порожні кутові дужки означають, що немає ніяких параметрів. А параметрів немає через те, що ми замінюємо єдиний параметр шаблону (T, який відповідає за тип даних) конкретним типом даних (bool). Потім ми пишемо ім’я класу і додаємо до нього <bool>, повідомляючи компілятору, що працюватимемо з типом bool.

Всі інші зміни — це просто деталі реалізації класу. Зверніть увагу, замість масиву з 8 змінних типу bool (8 байтів) ми використовуємо 1 змінну типу unsigned char (1 байт)..

Тепер при оголошенні об’єкту класу Repository8<T>, де T не є bool, ми отримаємо екземпляр загального шаблону класу, тоді як при оголошенні об’єкту Repository8<bool>, ми отримаємо екземпляр шаблону Repository8<bool>. Зверніть увагу, ми не змінювали інтерфейс класу, а залишили його відкритим (яким він і був), в той час як мова C++ надає нам можливість додавати/змінювати/видаляти методи класу. Справа в тому, що зміна інтерфейсу класу в спеціалізаціях шаблону не завжди вітається, тому що програміст може про це забути, а воно, в свою чергу, призведе до помилок.

Створимо об’єкти Repository8<T> і Repository8<bool>:

```
int main()
{
    // Оголошуємо цілочисельний об'єкт-масив (створюється екземпляр Repository8<T>, де T = int)
    Repository8<int> intRepository;

    for (int count = 0; count < 8; ++count)
        intRepository.set(count, count);

    for (int count = 0; count < 8; ++count)
        std::cout << intRepository.get(count) << '\n';

    // Оголошуємо об'єкт-масив типу bool (створюється екземпляр спеціалізації Repository8<bool>)
    Repository8<bool> boolRepository;

    for (int count = 0; count < 8; ++count)
        boolRepository.set(count, count % 5);

    for (int count = 0; count < 8; ++count)
        std::cout << (boolRepository.get(count) ? "true" : "false") << '\n';

    return 0;
}
```

Як ви можете бачити, результат той же, що і вище, де використовувався загальний шаблон класу Repository8:

```
0
1
2
3
4
5
6
7
false
true
true
true
true
false
true
true
```

Слід ще раз відзначити, що збереження відкритого інтерфейсу в шаблонах ваших класів разом зі спеціалізаціями, як правило, є хорошою ідеєю, оскільки це спрощує використання класів і їх логіку, хоч і не вважається строго необхідним.

Проблема

Розглянемо ще раз клас StaticArray:

```
template <class T, int size> // size є non-type параметром шаблону
class StaticArray
{
private:
	// Параметр size відповідає за довжину масиву
	T m_array[size];

public:
	T* getArray() { return m_array; }

	T& operator[](int index)
	{
		return m_array[index];
	}
};
```

Тут у нас є 2 параметри шаблону класу: параметр типу і параметр non-type.

Тепер припустимо, що нам потрібно написати функцію для виводу всіх елементів масиву. Хоча ми можемо зробити це через метод класу, ми реалізуємо це через окрему функцію (заради кращого розуміння теми).

Використовуючи шаблон функції, ми можемо написати наступне:

```
template <typename T, int size>
void print(StaticArray<T, size>& array)
{
	for (int count = 0; count < size; ++count)
		std::cout << array[count] << ' ';
}
```

Це дозволить нам зробити:

```
#include <iostream>
#include <cstring>

template <class T, int size> // size є non-type параметром шаблону
class StaticArray
{
private:
	// Параметр size відповідає за довжину масиву
	T m_array[size];

public:
	T* getArray() { return m_array; }

	T& operator[](int index)
	{
		return m_array[index];
	}
};

template <typename T, int size>
void print(StaticArray<T, size>& array)
{
	for (int count = 0; count < size; ++count)
		std::cout << array[count] << ' ';
}

int main()
{
	// Оголошуємо цілочисельний масив
	StaticArray<int, 5> int5;
	int5[0] = 0;
	int5[1] = 1;
	int5[2] = 2;
	int5[3] = 3;
	int5[4] = 4;

	// Виводимо елементи масиву
	print(int5);

	return 0;
}
```

І отримати:

0 1 2 3 4

Хоча все працює правильно, але є один нюанс. Розглянемо наступний код функції main():

```
int main()
{
	// Оголошуємо масив типу char
	StaticArray<char, 14> char14;

	strcpy_s(char14.getArray(), 14, "Hello, world!");

	// Виводимо елементи масиву
	print(char14);

	return 0;
}
```

Примітка:

Програма скомпілюється з наступним результатом:

H e l l o , w o r l d !

Для всіх типів, крім char, є сенс вказувати пробіл між кожним елементом масиву, щоб елементи не «злипалися». Однак з типом char є сенс вивести все разом, як рядок C-style, щоб не було зайвих пробілів. Як ми можемо це виправити?

Повна спеціалізація шаблону — рішення ?

Спочатку ми могли б подумати про використання спеціалізації шаблону функції. Однак проблема з повною спеціалізацією шаблону полягає в тому, що всі параметри шаблону повинні бути явно визначені. Наприклад :

```
#include <iostream>
#include <cstring>

	template <class T, int size> // size є non-type параметром шаблону
class StaticArray
{
private:
	// Параметр size відповідає за довжину масиву
	T m_array[size];

public:
	T* getArray() { return m_array; }

	T& operator[](int index)
	{
		return m_array[index];
	}
};

template <typename T, int size>
void print(StaticArray<T, size>& array)
{
	for (int count = 0; count < size; ++count)
		std::cout << array[count] << ' ';
}

// Шаблон функції print() з повною спеціалізацією шаблону класу StaticArray для роботи з типом
char і довжиною масиву 14
template <>
void print(StaticArray<char, 14>& array)
{
	for (int count = 0; count < 14; ++count)
		std::cout << array[count];
}

int main()
{
	// Оголошуємо масив типу char
	StaticArray<char, 14> char14;

	strcpy_s(char14.getArray(), 14, "Hello, world!");

	// Виводимо елементи масиву
	print(char14);

	return 0;
}
```

Як ви можете бачити, ми додали шаблон функції print() для роботи з типом char.

Результат:

```
Hello, world!
```

Хоча одна проблема вирішена, виникає інша проблема : використання повної спеціалізації шаблону класу означає, що ми повинні явно вказувати довжину переданого масиву! Розглянемо наступний приклад :

```
int main()
{
	// Оголошуємо масив типу char
	StaticArray<char, 12> char12;

	strcpy_s(char12.getArray(), 12, "Hello, dad!");

	// Виводимо елементи масиву
	print(char12);

	return 0;
}
```

Виклик print(char12) викличе шаблон функції print() із загальним шаблоном StaticArray<T, size>, тому що char12 є типу StaticArray<char, 12>, а шаблон функції print() приймає тільки StaticArray<char, 14>(довжина масиву відрізняється).

Хоча ми могли б скопіювати ще раз шаблон функції print() для роботи зі StaticArray<char, 12>, але це неефективно.А що, якщо нам потрібно буде пізніше використати масив з 5 або 20 елементами ? Знову копіювати шаблон ? Це зайва робота.

Очевидно, що повна спеціалізація шаблону класу тут є “рішенням - костилем”.Часткова спеціалізація шаблону — ось, що нам потрібно.

Часткова спеціалізація шаблону

Часткова спеціалізація шаблону дозволяє виконати спеціалізацію шаблону класу(але не функції!), де деякі(але не всі) параметри шаблону явно визначені.Для нашої вищенаведеної задачі ідеальне рішення полягає в тому, щоб шаблон функції print() працював зі StaticArray типу char, але при цьому розмір масиву не був фіксованим значенням, а міг варіюватися.

Ось наш шаблон функції print(), який приймає частково спеціалізований шаблон класу StaticArray :

```
// Шаблон функції print() з частково спеціалізованим шаблоном класу StaticArray<char, size> в якості параметра
template <int size> // size як і раніше є non-type параметром
void print(StaticArray<char, size>& array) // ми тут явно вказуємо тип char
{
	for (int count = 0; count < size; ++count)
		std::cout << array[count];
}
```

Як ви можете бачити, ми тут явно вказали тип char, але size залишили не фіксованим, тому функція print() працюватиме з масивами типу char будь - якого розміру.От і все!

Повний код програми :

```
#include <iostream>
#include <cstring>

template <class T, int size> // size є non-type параметром шаблону
class StaticArray
{
private:
	// Параметр size відповідає за довжину масиву
	T m_array[size];

public:
	T* getArray() { return m_array; }

	T& operator[](int index)
	{
		return m_array[index];
	}
```

```
};

template <typename T, int size>
void print(StaticArray<T, size>& array)
{
	for (int count = 0; count < size; ++count)
		std::cout << array[count] << ' ';
}

// Шаблон функції print() з частково спеціалізованим шаблоном класу StaticArray<char, size> в якості параметра
template <int size>
void print(StaticArray<char, size>& array)
{
	for (int count = 0; count < size; ++count)
		std::cout << array[count];
}

int main()
{
	// Оголошуємо масив типу char довжиною 14
	StaticArray<char, 14> char14;

	strcpy_s(char14.getArray(), 14, "Hello, world!");

	// Виводимо елементи масиву
	print(char14);

	// Тепер оголошуємо масив типу char довжиною 12
	StaticArray<char, 12> char12;

	strcpy_s(char12.getArray(), 12, "Hello, dad!");

	// Виводимо елементи масиву
	print(char12);

	return 0;
}
```

Результат:

```
Hello, world!Hello, dad!
```

Як і очікували.

Зверніть увагу, починаючи з C++14 часткова спеціалізація шаблону може використовуватися тільки з класами, але не з окремими функціями(для функцій використовується тільки повна спеціалізація шаблону).Наш приклад void print(StaticArray<char, size>& array) працює тільки тому, що шаблон функції print() приймає в якості параметра шаблон класу, який, в свою чергу, частково спеціалізований.

Часткова спеціалізація шаблонів методів

Обмеження часткової спеціалізації для функцій може призвести до певних проблем при роботі з методами класу. Наприклад, що, якби ми визначили StaticArray наступним чином :

```
template <class T, int size> // size є non-type параметром шаблону
class StaticArray
{
private:
	// Параметр size відповідає за довжину масиву
	T m_array[size];

public:
	T* getArray() { return m_array; }

	T& operator[](int index)
	{
		return m_array[index];
	}
```

```
	void print()
	{
		for (int i = 0; i < size; i++)
			std::cout << m_array[i] << ' ';
		std::cout << "\n";
	}
};
```

Функція print() є методом класу StaticArray<T, int>.Що станеться, якщо ми захочемо частково спеціалізувати шаблон функції print(), щоб метод працював по - іншому ? Ми можемо спробувати виконати наступне :

```
// Не спрацює
template <int size>
void StaticArray<double, size>::print()
{
	for (int i = 0; i < size; i++)
		std::cout << std::scientific << m_array[i] << " ";
	std::cout << "\n";
}
```

На жаль, це не спрацює, тому що ми намагаємося частково спеціалізувати шаблон функції, що робити заборонено.

Як же це можна обійти ? Одним з очевидних рішень є часткова спеціалізація шаблону всього класу :

```
#include <iostream>

template <class T, int size> // size є non-type параметром шаблону
class StaticArray
{
private:
	// Параметр size відповідає за довжину масиву
	T m_array[size];

public:
	T* getArray() { return m_array; }

	T& operator[](int index)
	{
		return m_array[index];
	}
	void print()
	{
		for (int i = 0; i < size; i++)
			std::cout << m_array[i] << ' ';
		std::cout << "\n";
	}
};

template <int size> // size є non-type параметром шаблону
class StaticArray<double, size>
{
private:
	// Параметр size відповідає за довжину масиву
	double m_array[size];

public:
	double* getArray() { return m_array; }

	double& operator[](int index)
	{
		return m_array[index];
	}
	void print()
	{
		for (int i = 0; i < size; i++)
			std::cout << std::scientific << m_array[i] << ' ';
		std::cout << "\n";
```

```
	}
};

int main()
{
	// Оголошуємо цілочисельний масив довжиною 5
	StaticArray<int, 5> intArray;

	// Заповнюємо масив, а потім виводимо його на екран
	for (int count = 0; count < 5; ++count)
		intArray[count] = count;
	intArray.print();

	// Оголошуємо масив типу double довжиною 4
	StaticArray<double, 4> doubleArray;

	for (int count = 0; count < 4; ++count)
		doubleArray[count] = (4.0 + 0.1 * count);
	doubleArray.print();

	return 0;
}
```

Результат:

```
0 1 2 3 4
4.000000e+00 4.100000e+00 4.200000e+00 4.300000e+00
```

Хоча це працює, але це не найкращий варіант, тому що у нас тепер купа дубльованого коду з StaticArray<T, size> в StaticArray<double, size>.

От якби можна було б використати код з StaticArray<T, size> в StaticArray<double, size> без дублювання. Нічого вам це не нагадує ? Як на мене, то це звучить, як відмінний варіант для застосування спадкування!

Ви можете почати з :

```
template <int size> // size є non-type параметром шаблону
class StaticArray<double, size> : public StaticArray < // а потім що?
```

Але як ми можемо посилатися на StaticArray ? Ніяк, але, на щастя, є обхідний шлях з використанням загального батьківського класу :

```
#include <iostream>

template <class T, int size> // size є non-type параметром шаблону
class StaticArray_Base
{
protected:
	// Параметр size відповідає за довжину масиву
	T m_array[size];

public:
	T* getArray() { return m_array; }

	T& operator[](int index)
	{
		return m_array[index];
	}
	virtual void print()
	{
		for (int i = 0; i < size; i++)
			std::cout << m_array[i] << ' ';
		std::cout << "\n";
	}
};

template <class T, int size> // size є non-type параметром шаблону
class StaticArray : public StaticArray_Base<T, size>
{
```

```
public:
	StaticArray()
	{

	}
};

template <int size> // size є non-type параметром шаблону
class StaticArray<double, size> : public StaticArray_Base<double, size>
{
public:

	virtual void print() override
	{
		for (int i = 0; i < size; i++)
			std::cout << std::scientific << this->m_array[i] << " ";

		std::cout << "\n";
	}
};

int main()
{
	// Оголошуємо цілочисельний масив довжиною 5
	StaticArray<int, 5> intArray;

	// Заповнюємо його, а потім виводимо на екран
	for (int count = 0; count < 5; ++count)
		intArray[count] = count;
	intArray.print();

	// Оголошуємо масив типу double довжиною 4
	StaticArray<double, 4> doubleArray;

	// Заповнюємо його, а потім виводимо на екран
	for (int count = 0; count < 4; ++count)
		doubleArray[count] = (4. + 0.1 * count);
	doubleArray.print();

	return 0;
}
```

Результат той же, що і в прикладі, наведеному вище, але дубльованого коду менше.

клас Repository :

```
#include <iostream>

template <class T>
class Repository
{
private:
	T m_value;
public:
	Repository(T value)
	{
		m_value = value;
	}

	~Repository()
	{
	}

	void print()
	{
		std::cout << m_value << '\n';
	}
};
```

Ми говорили про проблему цього шаблону при роботі з типом char*, коли виконувалося поверхневе копіювання(присвоювання вказівника) в конструкторі класу Repository.В якості рішення ми використовували повну спеціалізацію шаблону для створення спеціалізованої версії конструктора класу

Repository для роботи з типом char*, в якому виділялася пам’ять і виконувалося глибоке копіювання m_value.Ось спеціалізація конструктора і деструктора класу Repository для роботи з типом char* (з матеріалів того ж уроку) :

```
	template <>
Repository<char*>::Repository(char* value)
{
	// Визначаємо довжину value
	int length = 0;
	while (value[length] != '\0')
		++length;
	++length; // +1, враховуючи нуль-термінатор

	// Виділяємо пам'ять для зберігання значення value
	m_value = new char[length];

	// Копіюємо фактичне значення з value в m_value
	for (int count = 0; count < length; ++count)
		m_value[count] = value[count];
}

template<>
Repository<char*>::~Repository()
{
	delete[] m_value;
}
```

Хоча все відмінно працює з типом char*, але як щодо інших типів вказівників(наприклад, int*) ? Оскільки T — це будь - який тип вказівника, то при роботі з тим же int* виконається поверхневе копіювання(що нам не потрібно), або нам доведеться дублювати вищенаведений код(спеціалізація конструктора і деструктора), але вже замість char* використовувати int* .А дублювання коду, як ми вже знаємо, не найкращий варіант!

На щастя, використовуючи часткову спеціалізацію шаблону, ми можемо визначити спеціальну версію класу Repository, яка працювала б з усіма типами вказівників(при цьому не потрібно вказувати конкретні типи вказівників) :

```
#include <iostream>

	// Загальний шаблон класу Repository
	template <class T>
class Repository
{
private:
	T m_value;
public:
	Repository(T value)
	{
		m_value = value;
	}

	~Repository()
	{
	}

	void print()
	{
		std::cout << m_value << '\n';
	}
};

template <typename T>
class Repository<T*> // часткова спеціалізація шаблону класу Repository для роботи з типами вказівників
{
private:
	T* m_value;
public:
	Repository(T* value) // T - тип вказівника
```

```
	{
		// Виконуємо глибоке копіювання
		m_value = new T(*value); // тут копіюється тільки одне окреме значення (не масив значень)
	}

	~Repository()
	{
		delete m_value; // а тут виконується видалення цього значення
	}

	void print()
	{
		std::cout << *m_value << '\n';
	}
};
```

І приклад з практики :

```
int main()
{
	// Оголошуємо цілочисельний об'єкт для перевірки роботи загального шаблону класу
	Repository<int> myint(6);
	myint.print();

	// Оголошуємо об'єкт з типом вказівник для перевірки роботи часткової спеціалізації шаблону класу
	int x = 8;
	Repository<int*> myintptr(&x);

	// Якби в myintptr виконалося поверхневе копіювання (присвоювання вказівника),
	// то зміна значення x призвела б до зміни значення myintptr
	x = 10;
	myintptr.print();

	return 0;
}
```

Результат:

```
6
8
```

При оголошенні об’єкта myintptr з типом int*, компілятор бачить, що ми раніше визначили часткову спеціалізацію шаблону класу для роботи з типами вказівників, і, враховуючи, що ми використовували тип int*, компілятор створить екземпляр часткової спеціалізації шаблону для роботи з типом вказівника.Конструктор цієї спеціалізації виконує глибоке копіювання параметру x.Пізніше, коли ми змінюємо значення x на 10, myintptr.m_value ніяк не змінюється, тому що виконалося глибоке копіювання, при якому m_value отримав свою власну копію x.

Якби цієї часткової спеціалізації не існувало, то створився б екземпляр загального шаблону класу, в якому виконалося б поверхневе копіювання, а myintptr.m_value і x вказували б на одну і ту ж адресу в пам’яті.В такому випадку, при зміні значення змінної x на 10, ми також зачепили б і значення myintptr(воно також стало б дорівнювати 10).

Варто відзначити, що, оскільки в нашій частковій спеціалізації копіюється лише одне значення, при роботі з рядками C - style копіюватиметься тільки перший символ(тому що рядок — це масив, а вказівник на масив вказує тільки на перший елемент масиву).Якщо ж потрібно повністю скопіювати рядок, то спеціалізація конструктора(і деструктора) для типу char* повинна бути повною.В такому випадку, повна спеціалізація матиме більший пріоритет, ніж часткова спеціалізація.Наприклад, ось програма, в якій використовується як часткова спеціалізація для роботи з типами вказівників, так і повна спеціалізація для роботи з типом char* :

```
#include <iostream>
#include <cstring>

// Загальний шаблон класу Repository для роботи з НЕ вказівниками
template <class T>
class Repository
```

```
{
private:
	T m_value;
public:
	Repository(T value)
	{
		m_value = value;
	}

	~Repository()
	{
	}

	void print()
	{
		std::cout << m_value << '\n';
	}
};

// Часткова спеціалізація шаблону класу Repository для роботи з вказівниками
template <class T>
class Repository<T*>
{
private:
	T* m_value;
public:
	Repository(T* value)
	{
		m_value = new T(*value);
	}

	~Repository()
	{
		delete m_value;
	}

	void print()
	{
		std::cout << *m_value << '\n';
	}
};

// Повна спеціалізація шаблону конструктора класу Repository для роботи з типом char*
template <>
Repository<char*>::Repository(char* value)
{
	// Визначаємо довжину value
	int length = 0;
	while (value[length] != '\0')
		++length;
	++length; // +1, враховуючи нуль-термінатор

	// Виділяємо пам'ять для зберігання значення value
	m_value = new char[length];

	// Копіюємо фактичне значення value в m_value
	for (int count = 0; count < length; ++count)
		m_value[count] = value[count];
}

// Повна спеціалізація шаблону деструктора класу Repository для роботи з типом char*
template<>
Repository<char*>::~Repository()
{
	delete[] m_value;
}

// Повна спеціалізація шаблону метода print() для роботи з типом char*.
// Без цього вивід Repository<char*> призвів би до виклику Repository<T*>::print(), який
виводить тільки одне значення (у випадку з рядком C-style – тільки перший символ)
template<>
```

```
void Repository<char*>::print()
{
	std::cout << m_value;
}

int main()
{
	// Оголошуємо цілочисельний об'єкт для перевірки роботи загального шаблону класу
	Repository<int> myint(6);
	myint.print();

	// Оголошуємо об'єкт з типом вказівник для перевірки роботи часткової спеціалізації
шаблону
	int x = 8;
	Repository<int*> myintptr(&x);

	// Якби в myintptr виконалося поверхневе копіювання (присвоювання вказівника),
	// то зміна значення x призвела б до зміни значення myintptr
	x = 10;
	myintptr.print();

	// Динамічно виділяємо тимчасовий рядок
	char* name = new char[40]{ "Anton" }; // необхідна підтримка C++14

	// Якщо ваш компілятор не підтримує C++14, то закоментуйте рядок вище і розкоментуйте
рядки нижче
//	char *name = new char[40];
//	strcpy(name, "Anton");

	// Зберігаємо ім'я
	Repository<char*> myname(name);

	// Видаляємо тимчасовий рядок
	delete[] name;

	// Виводимо ім'я
	myname.print();
}
Все працює як потрібно :

6
8
Anton
```

Загалом, використання часткової спеціалізації шаблону класу для роботи з типами вказівників дозволяє передбачити всі можливі варіанти використання коду на практиці.