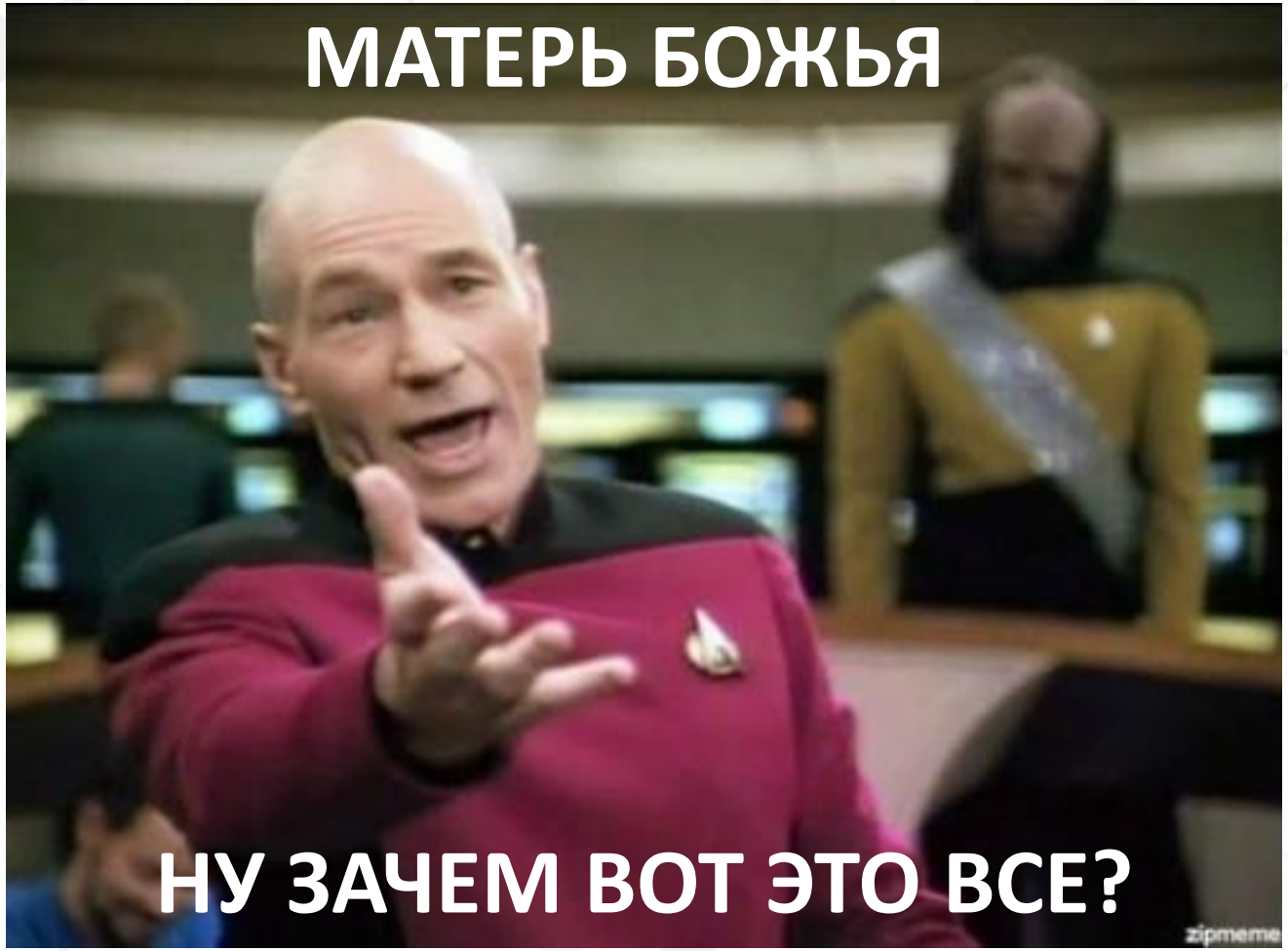
Язык программирования C++

Шаблоны.

Преподаватели:
Пысин Максим Дмитриевич, ассистент кафедры ИКТ
Краснов Дмитрий Олегович, аспирант кафедры ИКТ
Лобанов Алексей Владимирович, аспирант кафедры ИКТ
Крашенинников Роман Сергеевич, аспирант кафедры ИКТ




```
int main(){
    string text;
    getline(cin, text);
    cout << text << endl;
    cout << text.length() << endl;
    map<char, int> letters;
    for(auto letter: text){
        if(letters.cc
            letters[]
    const int N = 10;
    std::array<int, 3> a1 { {1, 2, 3} };
    std::array<int, 3> a2 = {1, 2, 3}; //
    std::array<int, N> arrayCPP = {0};
    for(int el: arrayCPP)
        cout << el << ": "; // 0: 0: 0: 0
```

```
    srand(time(0));
    const int N = 5;
    vector<int> v(N);
    cout << v.size() <
    for(int i = 0; i <
```

```
    srand(time(0));
    const int N = 10;
    list<int> list;
    cout << "Empty: " << l
    for(int i = 0; i < N;
```

Пример, перегрузка

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

int main(){
    int i = maximum(13, 10);
    cout << i << endl; // 13

    double d = maximum(0.97, 1.76);
    cout << d << endl; // 1.76

    char s = maximum('a', '1');
    cout << s << endl; // a
}
```

Метапрограммирование

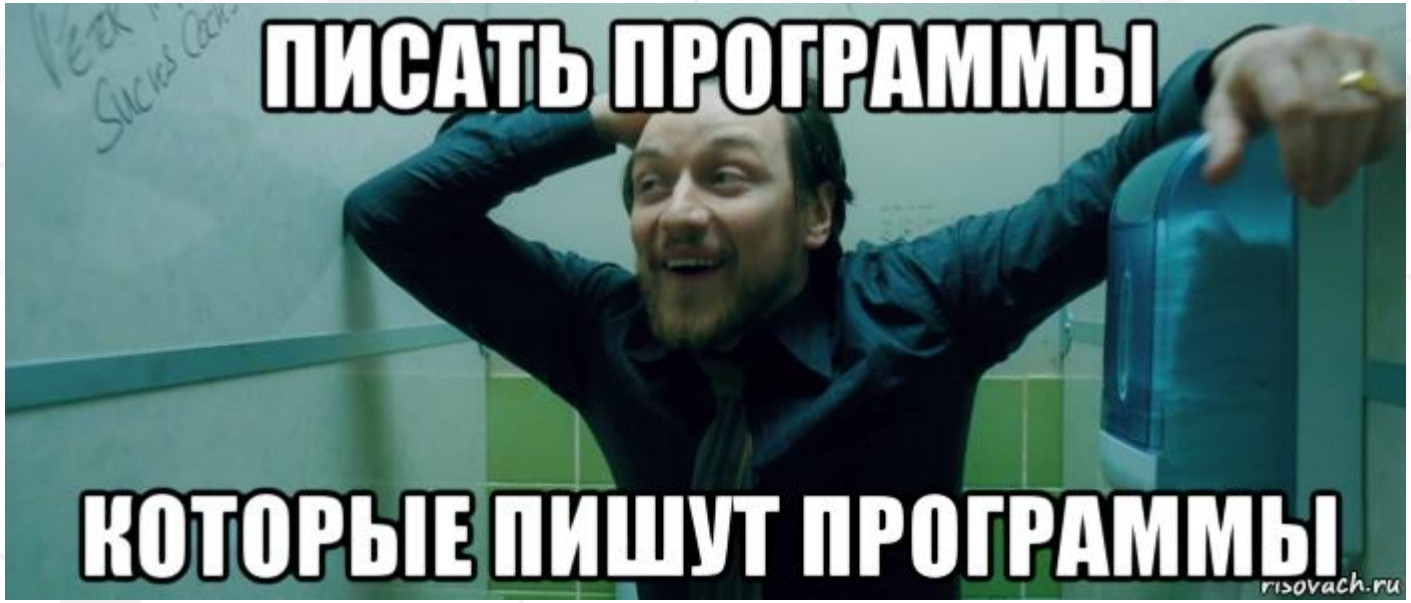
Метапрограммирование — вид программирования, связанный с созданием программ, которые порождают другие программы как результат своей работы

При этом подходе код программы не пишется вручную, а создаётся автоматически программой-генератором на основе другой программы.

Различаются два принципиально различных вида кодогенерации:

генератор является физически отдельной бинарной программой, необязательно написанной на целевом языке.

целевой язык является одновременно языком реализации генератора, так что метапрограмма составляет с целевой программой единое целое.



Шаблоны

Шаблоны (**template**) — средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).

Шаблоны функций *– это обобщенное описание поведения функций, которые могут вызываться для объектов разных типов.* Другими словами, шаблон функции (шаблонная функция, обобщённая функция) представляет собой семейство разных функций (или описание алгоритма). По описанию шаблон функции похож на обычную функцию: разница в том, что некоторые элементы не определены (типы, константы) и являются параметризованными.

Шаблоны классов *– обобщенное описание пользовательского типа, в котором могут быть параметризованны атрибуты и операции типа.* Представляют собой конструкции, по которым могут быть сгенерированы действительные классы путём подстановки вместо параметров конкретных аргументов.

Первый способ: шаблонная функция

Шаблоны объявляются при помощи ключевого слова `template`, которое является началом шаблона, и сигнализирует компилятору интерпретировать дальнейшую функцию как шаблон. Далее следуют угловые скобки `< >` между которыми через запятую указываться параметры шаблона. Основными параметрами шаблона являются вариативные типы, для того что бы их указать нужно использовать либо ключевое слово `typename` либо `class` разницы между которыми в данном контексте нет никакой, и можно использовать любое, в итоге сигнатура шаблонной функции будет такой:

```
template <typename <my type name>>
const <my type name>& <function name>(const <my type name>& a, const <my type name>& b){
    return a > b ? a: b;
}
```

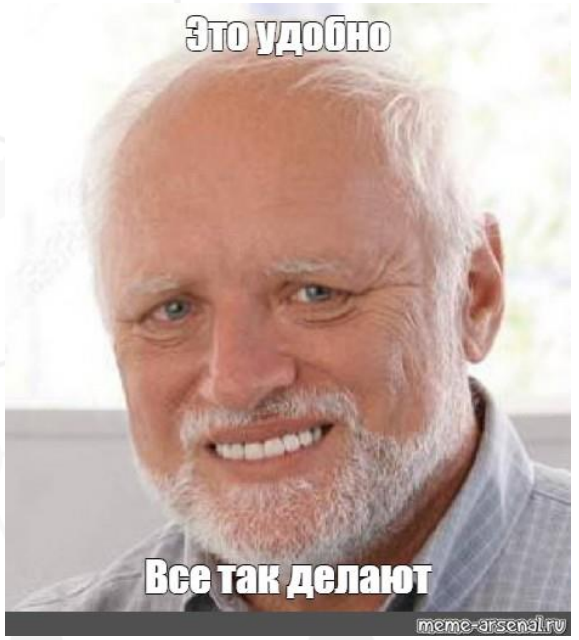
После объявления шаблона, компилятор соберет все места использования вашей шаблонной функции и сгенерирует то количество уникальных функций каждая их которых будет принимать в себя аргументы определенных типов, сколько различающихся по типу посылаемых значений вызовов будет в вашем коде.

У шаблонов функций есть два основных недостатка:

- Сумасшедшие сообщения об ошибках, которые намного сложнее расшифровать
- увеличивают время компиляции и размер кода

Пример, шаблонная функции

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

int main(){
    int i = maximum(13, 10);
    cout << i << endl; // 13

    double d = maximum(0.97, 1.76);
    cout << d << endl; // 1.76

    char s = maximum('a', '1');
    cout << s << endl; // a
}
```



Вся задача по перегрузки функции ложиться на компилятор и функция может быть перегружена только в том случае, если тип имеет соответствующие операции сравнения.

Первый способ: auto

Если вы вспомните существует такой специализированный тип в C++ как auto который позволяет не указывать конкретный тип, а компилятор самостоятельно подберет нужный во время компиляции и сделает все необходимое для работоспособности вашего кода, поэтому функцию можно так же переделать так:

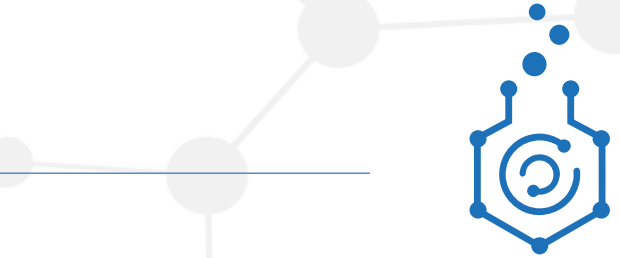
```
const auto& maximum(const auto& a, const auto& b){
    return a > b ? a: b;
}
```

!Важно отметить что это работает не на любом компиляторе, да и редактор на такое использование в большинстве случаев будет ругаться, но на mingw оно копилируется.

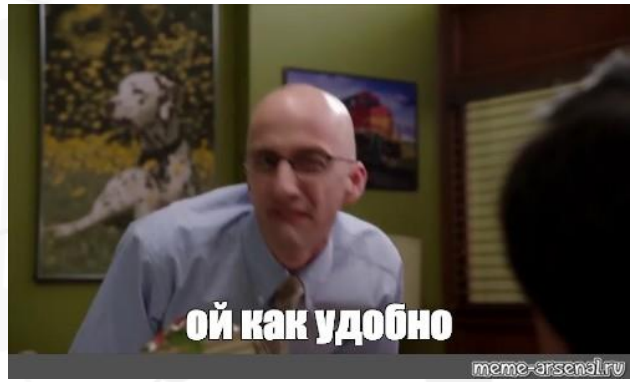
Отличие шаблонов от подобного рода созданных функций в том, что в шаблоне мы не указываем точный тип, но мы требуем что бы аргументы в функции были одного типа, и возвращаемое значение было этого же типа, тогда как auto не предполагает никакого требования к вызову такой функции, почему и считается неприменимым к подобному использованию.

Любой шаблон функции предполагает наличие определенных свойств параметризованного типа, в зависимости от реализации (например, оператора копирования, оператора сравнения, наличия определенного метода и т.д.).

Пример, авто определение типа



```
const auto& maximum(const auto& a, const auto& b){
    return a > b ? a: b;
}

int main(){
    int i = maximum(13, 10);
    cout << i << endl; // 13

    double d = maximum(0.97, 1.76);
    cout << d << endl; // 1.76

    char s = maximum('a', '1');
    cout << s << endl; // a
}
```



Такой способ появился после 11 стандарта. По сути, вся задача по перегрузки функции ложиться на компилятор и функция может быть перегружена только в том случае, если тип имеет соответствующие операции сравнения.

Пример, шаблонная функции с собственным классом

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

class MyCombinedString{
    private:
        string s;
    public:
        MyCombinedString(string s): s(s){}
    friend bool operator>(
        const MyCombinedString& msa, const MyCombinedString& msb
    );
    friend ostream& operator<<(
        ostream& out, const MyCombinedString& ms
    );
};
bool operator>(
    const MyCombinedString& msa, const MyCombinedString& msb
){
    return msa.s.length() > msb.s.length();
}
ostream& operator<<(ostream& out, const MyCombinedString& ms){
    return out << ms.s;
}
```

```
int main(){
    int i = maximum(13, 10);
    cout << i << endl; // 13

    double d = maximum(0.97, 1.76);
    cout << d << endl; // 1.76

    char s = maximum('a', '1');
    cout << s << endl; //a

    MyCombinedString ms = maximum(
        MyCombinedString("12.1234"),
        MyCombinedString("12")
        );
    cout << ms << endl; // 12.1234
}
```

Шаблонная функция и собственная реализация

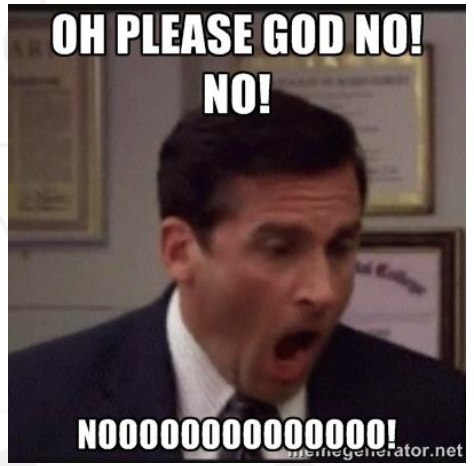
В некоторых случаях шаблон функции является неэффективным или неправильным для определенного типа. В этом случае можно специализировать шаблон, — то есть написать реализацию для данного типа.

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

template <>
const string& maximum(const string& a,
const string& b){
    return a.length() > b.length() ? a :
b;
}
```

```
int main(){
    int i = maximum(13, 10);
    cout << i << endl; // 13

    double d = maximum(0.97, 1.76);
    cout << d << endl; // 1.76

    char s = maximum('a', '1');
    cout << s << endl; // a

    string ms = maximum(string("11.1234"),
string("12"));
    cout << ms << endl; // 11.1234
}
```

Шаблонный класс

Шаблоны классов аналогично шаблону функции объявляются при помощи ключевого слова template. Далее следуют угловые скобки < > между которыми через запятую указываться параметры шаблона. После идет стандартное описание класса в котором для каких либо целей могут использоваться параметры шаблона, итоговая сигнатура будет выглядеть так:




```
template <typename <имя аргумента>>
class <имя класса> {
    private:
        <имя аргумента>* <имя атрибута>;
        …
    public:
        <имя аргумента> <имя метода> (…){…}
        …
        <имя аргумента> <имя метода> (…); !!!! Вынос методов в cpp имеет нюансы
        template <typename <другое имя аргумента>>
        <другое имя аргумента> <имя дружественного метода> (<другое имя аргумента>, …);

};

template <typename <имя аргумента>> !!!! Вынос методов в cpp имеет нюансы
<имя аргумента> <имя класса><<имя аргумента>>:<имя метода> (…) {…}

template <typename <другое имя аргумента>>
<другое имя аргумента> <имя дружественного метода> (<другое имя аргумента>, …){…}
```

Больше фактов о шаблонах

Тип описанный в шаблоне может использоваться с префиксом const, а так же может использоваться как указательный или ссылочный тип, без изменения шаблона, т.е. если есть шаблон с типом T, то в пределах шаблонной функции или класса могут быть объявлены константы типа T, а так же переменные типа указатель на T(T*) и ссылка на T(T&).

```
template <typename T>
T* maximum(const T& a, const T* b)
```

Для шаблонного класса, если используется дружественные функции, при написании определений метода требуется для каждого метода использовать полное объявление шаблона:

```
template <typename T>
class Array{
    public:
        T& operator[](int i);
        friend ostream& operator<<(ostream& out, Array<TF>& arr);
};

template <typename T>
T& Array<T>::operator[](int i) {
    // Описание
}
```

Пример, шаблонный класс

```
template <typename T>
class Array{
    private:
        T* container = nullptr;
        int length = 0;
    public:
        Array(const int N, T nuller = 0):
            length(N), container(new T[N])
        {
            for(int i = 0; i < length; i++)
                container[i] = nuller;
        }
        T& operator[](int i) {
            return container[i];
        }
        getLen() {
            return length;
        }
        template <typename TF>
        friend ostream& operator<<(ostream& out, Array<TF>& arr);
};
```

```
void unite(Array& second){
    T* result = new T[length + second.length];
    int k = 0;
    for(int i = 0; i < length; i++){
        result[k++] = container[i];
    }
    for(int i = 0; i < second.length; i++)
        result[k++] = second[i];
    delete container;
    container = result;
    length += second.length;
}
```

Пример, шаблонный класс

```
template <typename T>
class Array{…
    public:
        class Iterator{
            private:
                Array<T>* container;
                int index;
            public:
                Iterator(int index, Array<T>* container):
                    container(container),
                    index(index > container->length ? -1: index){
                }
                Iterator& operator++(){
                    if(index != container->length)
                        index++;
                    else
                        index = -1;
                    return *this;
                }
                T& operator*(){
                    return (*container)[index];
                }
                operator!=(Iterator& it){
                    return it.index != index;
                }
        };
};
```


```
template <typename T>
ostream& operator<<(ostream& out, Array<T>& arr){
    out << "[ ";
    for(int i = 0; i < arr.length; i++)
        if (i != arr.length - 1)
            out << arr[i] << ", ";
        else
            out << arr[i];
    out << " ]";
    return out;
}
```

Пример, шаблонный класс

```
int main(){
    srand(time(0));
    Array<int> first(10);
    Array<int> second(5);
    Array<double> third(6, 0.5);
    for(int i = 0; i < second.getLen(); i++)
        second[i] = rand() % 100;
    for(auto el: first)
        cout << el << endl;  // [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
    cout << second << endl; // [ 45, 78, 71, 67, 30 ]
    cout << third << endl; // [ 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 ]
    first.unite(second);
    cout << first << endl; // [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 30, 38, 91, 22, 34 ]
    // first.unite(third); // не существует подходящего определяемого пользователем преобр
азования из "Array<double>" в "const Array<int>"
}
```

Разбиение шаблона на файлы

Для организации больших программ со значительным количеством классов и большими размерами этих классов, является логичным желание разбить программу на несколько файлов. Шаблоны так же можно выносить в отдельные от основной функции файлы, но у них существует ряд ограничений.

Во первых. Шаблон можно вынести в h файл из файла с функцией main или с тем местом где он используется, ничего не меняя в этом классе. Это является стандартным способом отделения шаблона от кода.

main.cpp ...\iterable M | array.cpp U | array.h ...\incpp U

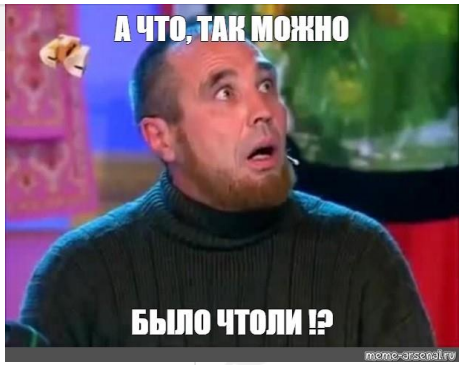
9 > class_template > iterable > main.cpp > main()

```
You, 6 минут назад | 1 author (You)
#include <iostream>
#include <ctime>
#include <string>

using std::cout;
using std::endl;
using std::ostream;

You, 6 минут назад | 1 author (You)
template <typename T>
class Array{
    private:
        T* container = nullptr;
        int length = 0;
    public:
        You, 6 минут назад | 1 author (You)
        class Iterator{
            private:
                Array<T>* container;
                int index;
            public:
                Iterator(int index, Array<T>* container):
                    container(container),
                    index(index > container->length ? -1: index){
                }
                Iterator& operator++(){
                    if(index != container->length - 1)
```

array.h U

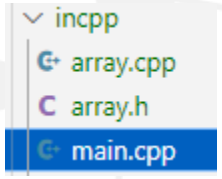
9 > class_template > iterable > inh > array.h > Array<T> > Iterator > container

```
#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>

template <typename T>
class Array{
    private:
        T* container = nullptr;
        int length = 0;
    public:
        class Iterator{
            private:
                Array<T>* container;
                int index;
            public:
                Iterator(int index, Array<T>* container):
                    container(container),
                    index(index > container->length ? -1: index){
                }
                Iterator& operator++(){
                    if(index != container->length - 1)
                        index++;
                    else
                        index = -1;
                    return *this;
                }
```



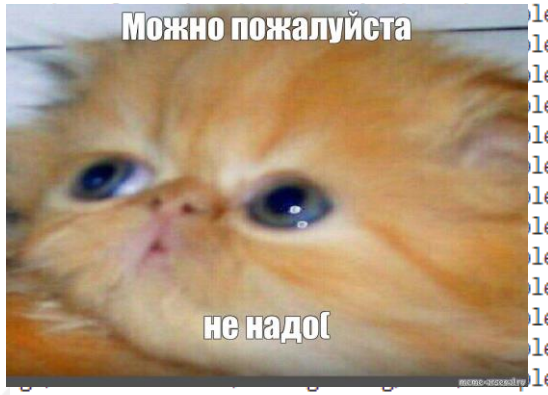
Разбиение шаблона на файлы

Во вторых. Можно пойти дальше и изъявить желание разбить шаблон на файлы h и cpp. Это желание столкнется с рядом проблем:
Сначала, требуется в каждом вынесенном методе использовать приставку template <typename T>.



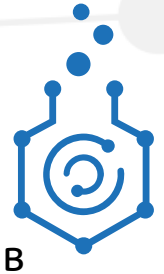

```
template <typename T>
int Array<T>::getLen() {
    return length;
}
```

Далее, шаблонный класс разделенный на 2 части после сборки на этапе единиц трансляции, начнет вызывать проблемы на этапе линковки:




```
C:\Users\root\AppData\Local\Temp\ccUvQna3.o: In function `main':
les/9/class_template/iterable/main.cpp:82: undefined reference to `Array<int>::Array(int, int)'
les/9/class_template/iterable/main.cpp:83: undefined reference to `Array<int>::Array(int, int)'
les/9/class_template/iterable/main.cpp:84: undefined reference to `Array<double>::Array(int, double)'
les/9/class_template/iterable/main.cpp:86: undefined reference to `Array<int>::operator[](int)'
les/9/class_template/iterable/main.cpp:87: undefined reference to `Array<int>::end()'
les/9/class_template/iterable/main.cpp:87: undefined reference to `Array<int>::Iterator::operator!=(Array<int>::Iterator&)'
les/9/class_template/iterable/main.cpp:87: undefined reference to `Array<int>::Iterator::operator*()'
les/9/class_template/iterable/main.cpp:87: undefined reference to `Array<int>::Iterator::operator++()'
les/9/class_template/iterable/main.cpp:89: undefined reference to `std::ostream& operator<< <int>(std::ostream&, Array<int>&)'
les/9/class_template/iterable/main.cpp:90: undefined reference to `std::ostream& operator<< <int>(std::ostream&, Array<int>&)'
les/9/class_template/iterable/main.cpp:91: undefined reference to `std::ostream& operator<< <double>(std::ostream&, Array<double>&)'
les/9/class_template/iterable/main.cpp:92: undefined reference to `Array<int>::unite(Array<int>&)'
les/9/class_template/iterable/main.cpp:93: undefined reference to `std::ostream& operator<< <int>(std::ostream&, Array<int>&)'
collect2.exe: error: ld returned 1 exit status
```

Разбиение шаблона на файлы



Для решения этой проблемы, потребуется в конце cpp файла с реализацией, указать компилятору какие в варианты шаблонов под какие типы нужно сгенерировать.


```
template class Array<int>;
template class Array<double>;
template std::ostream& operator<<(std::ostream& out, Array<int>& arr);
template std::ostream& operator<<(std::ostream& out, Array<double>& arr);
```

Обратите внимание, что если у класса есть дружественные функции, они должны быть так же обозначены, как варианты для обязательной компиляции.

Причина происходящего в том, что при разбиении на 2 cpp файла, компилятор, при компиляции файла с реализацией шаблонного класса не генерирует все варианты шаблонов. Тогда как, к моменту компиляции файла с main функцией нет ни одного варианта скомпилированного шаблона, и при этом нет самого шаблона. Поэтому возникает ошибка связывания при вызове любой функции шаблонного класса.

Дополнительные аргументы шаблона класса

Аргументы шаблонов могут быть в том числе и не типами, а конкретными значениями какого либо типа. Однако есть существенные ограничения:

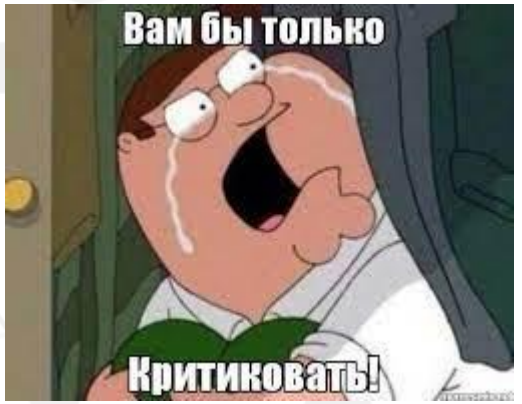
- В роли параметров шаблонов могут выступать целочисленные константы (включая перечисления) или указатели на объекты с внешним связыванием.
- Использование чисел с плавающей точкой и объектов с типом класса в качестве параметров шаблона не допускается.

```
template <typename T, int SIZE>
class Array{
    private:
        T* container = nullptr;
    public:
        Array(T nuller = 0): container(new T[SIZE]){
            for(int i = 0; i <  SIZE; i++)
                container[i] = nuller;
        }
        T& operator[](int i) {
            return container[i];
        }
};
```

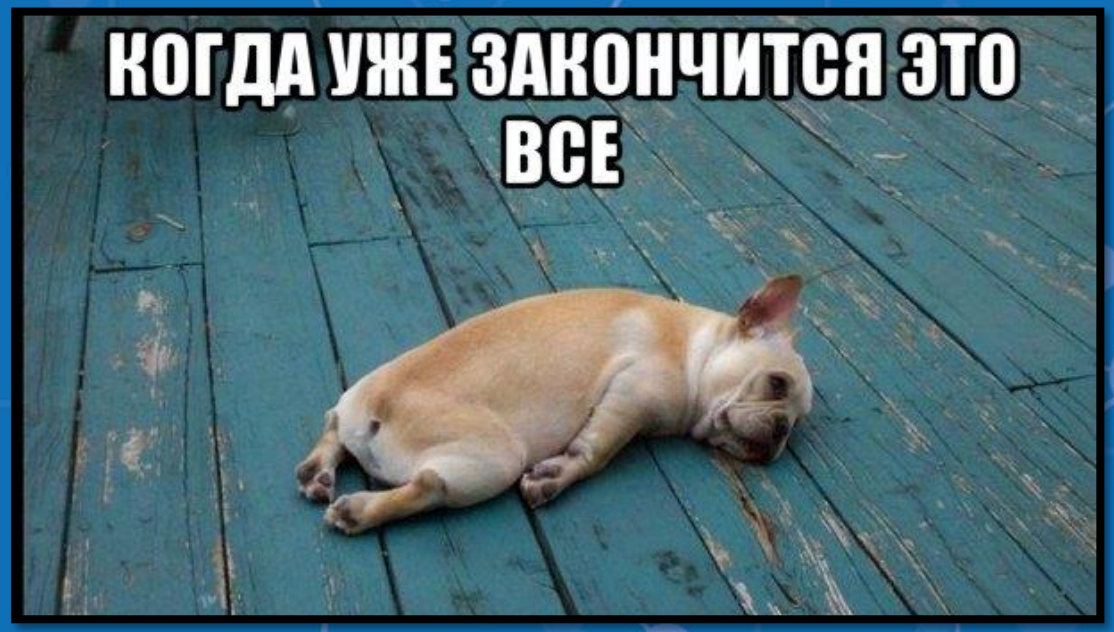
Немного критики

Было бы глупо отрицать проблемы шаблонов которые существуют, вот те которые выделяют и которые плавают на поверхности вопроса:

- Проблема переноса на другие платформы
- Проблема поддержки отладки
- Проблема поддержки ввода/вывода в процессе спецификации шаблона
- Длительное время компиляции
- Снижение читабельности кода
- Проблему диагностики ошибок из-за их сомнительного формата
- Проблема проверки аргументов шаблона (Решена при помощи концептов в C++ 20)



В некоторых языках есть аналоги шаблонов, которые по мнению некоторых работают лучше, или имеют другие преимущества, так Java реализует свою систему обобщенного программирования, есть языки которые специализируются на программировании типов, что требует от них обобщающих подходов, и при этом они не требуют шаблонов как в C++. Так же есть язык D, в котором систем шаблонов считается более мощной чем в C++.



Спасибо за внимание