

Метапрограммирование на C++

14 июля 2022 г.

Оглавление

1	Введение в шаблоны	3
1.1	Шаблоны функций	3
1.2	constexpr, constexpr, constexpr	12

Глава 1

Введение в шаблоны

Основное применение шаблонов берёт свои корни в необходимости обобщать алгоритмы и структуры данных на произвольные типы. Однако, как мы убедимся в рамках данного пособия, за годы их возможности и область применения вышли сильно за рамки этой изначальной мотивировки. В рамках данной главы мы познакомимся с этим базовым применением шаблонов.

1.1. Шаблоны функций

Суть этого понятия кроется в самом названии раздела. Шаблон функции – отрывок кода, не являющийся сам по себе функцией, но служащий шаблоном по которому компилятор будет создавать за нас функции. Предположим, что мы реализовали алгоритм двоичного поиска целого числа в массиве:

```
size_t binarySearch(std::span<int const> data, int value)
{
    size_t left = 0;
    size_t right = data.size();
    while (right - left > 1)
    {
        size_t middle = std::midpoint(left, right);
        if (data[middle] ≤ value)
        {
```

```

        left = middle;
    }
    else
    {
        right = middle;
    }
}
return left;
}

```

Ясно, что написанный нами код на самом деле не использует никаких уникальных для типа `int` свойств. Алгоритм подходит и для `long long`, и для `float`. Как добиться возможности использовать наш алгоритм и для других типов данных?

Плохой программист на языке C посоветует скопировать эту функцию несколько раз, заменяя `int` на другие интересующие нас типы данных, а в конец названия функции дописывая тип данных, с которым она работает. Недостаток этого подхода очевиден: дублирование кода. Более опытный программист на C порекомендует воспользоваться макросами во избежание дублирования. Такой подход используется, например, в функциях стандартной библиотеки C `fabs`, `fabsf` и `fabsl`. Однако мы на самом деле не решили основную проблему: мы не обобщили алгоритм, а сделали три разных алгоритма с разными названиями. Это не позволит нам использовать его в других обобщённых алгоритмах не продолжая дублировать код или использовать макросы. Любой другой алгоритм использующий наш двоичный поиск тоже будет должен предоставлять несколько разных версий с суффиксами названия, соответствующими типу данных. Более того, код для разных типов данных уже не может быть одинаковым, он обязан использовать разные версии функции бинарного поиска. Конечно же и эта проблема решается ловким использованием макросов. Но опасность излишнего использования макросов широко известна в среде системного программирования: макросы ничего не знают о синтаксисе языка, они лишь позволяют автоматизировать процесс копирования кода путём автоматических вставок. Неосторожное их использование может быстро привести код в состояние спагетти.

```

#define MAKE_BINARY_SEARCH(Type, Suffix)           \
size_t binarySearch ## Suffix(

```

```

    std::span<Type const> data, Type value)           \
{                                                       \
    size_t left = 0;                                   \
    size_t right = data.size();                         \
    while (right - left > 1)                             \
    {                                                   \
        size_t middle = std::midpoint(left, right); \
        if (data[middle] ≤ value)                     \
        {                                             \
            left = middle;                             \
        }                                             \
        else                                         \
        {                                             \
            right = middle;                           \
        }                                             \
    }                                                 \
    return left;                                       \
}
MAKE_BINARY_SEARCH(int, i)
MAKE_BINARY_SEARCH(float, f)
MAKE_BINARY_SEARCH(double, d)

```

Стоит отметить, что язык C++, в отличие от C, позволяет перегружать функции. Это решает проблему суффиксов, но не избавляет нас от необходимости использовать макросы.

Альтернативным подходом от мира C было бы использование указателя на функцию-компаратор:

```

size_t binarySearch(
    const void* dataStart, size_t elementSize,
    size_t elementCount, void* value,
    bool (*compare)(const void*, const void*))
{
    size_t left = 0;
    size_t right = elementCount;
    while (right - left > 1)
    {
        size_t middle = std::midpoint(left, right);
        const void* element =
            reinterpret_cast<const std::byte*>(dataStart)
            + middle * elementSize;
        if (compare(element, value))
        {

```

```

        left = middle;
    }
    else
    {
        right = middle;
    }
}
return left;
}

```

Недостаток такого подхода очевиден: необходима низкоуровневая работа с байтами (аналогично стандартной функции `qsort`), засоряющая логику нашего алгоритма. Есть у него и преимущество: пользователь может использовать алгоритм не только с predetermined набором типов данных, но и со своими структурами и классами. И наконец мы избавили себя от необходимости писать несколько версий одной и той же функции. Однако использование функции с таким интерфейсом нельзя назвать приятным, необходимо писать очень много служебного кода:

```

bool compareInt(const void* first, const void* second)
{
    return *reinterpret_cast<const int*>(first)
        ≤ *reinterpret_cast<const int*>(second);
}
...
std::vector<int> data = ...;
int target = 42;
binarySearch(data.data(), sizeof(int), data.size(),
             &target, &compareInt);

```

И наконец отметим, что мы пожертвовали производительностью в угоду общности. Предыдущий подход позволял компилятору превратить оператор сравнения чисел в одну ассемблерную инструкцию, а в текущем в качестве компаратора может быть указатель на любую функцию, заинлайнить которую вообще говоря может не получиться.

Похожий подход посоветовал бы программист на java. А именно, выделить интерфейс с необходимым для работы алгоритма функционалом и вместо конкретного типа принимать указатели на интерфейсы, для примитивных типов написать обёртки аналогичные имеющимся в языке java (`Integer`, `Float` и т.д.), и хранить всё на куче.

```
struct IComparable
{
    virtual bool isLessOrEqual(IComparable const * other) = 0;
}

size_t binarySearch(
    std::span<IComparable const *> data,
    IComparable const * value)
{
    size_t left = 0;
    size_t right = data.size();
    while (right - left > 1)
    {
        size_t middle = std::midpoint(left, right);
        if (data[middle]→isLessOrEqual(value))
        {
            left = middle;
        }
        else
        {
            right = middle;
        }
    }
    return left;
}
```

Этот подход достаточно похож на предыдущий, но нам больше не нужно заниматься низкоуровневыми манипуляциями байтами. Однако трейдофф состоит в производительности: мы больше не можем передавать на вход алгоритму обычный массив данных, необходимо передавать массив указателей на данные. Также вызов метода-компаратора ещё дороже чем вызов указателя на функцию-компаратор.

Итак, перечислим недостатки встречавшиеся нам в предыдущих подходах.

- Сложный пользовательский интерфейс
- Дублирование логики
- Игнорирование синтаксиса языка (макросы)
- Протекание абстракций (необходимость работать с байтами)

- Необходимость в индерекции данных (интерфейсы)
- Необходимость в индерекции функций (компараторы)

К счастью, язык C++ предоставляет нам инструмент, позволяющий избежать каждой из этих проблем: шаблоны. Больше всего они похожи на первый рассмотренный нами подход с макросами. Шаблон – аналог макроса, учитывающий синтаксис языка ещё до подстановки аргументов. Синтаксис шаблона функции выглядит следующим образом.

```
template<class T>
size_t binarySearch(std::span<T const> data, T value)
{
    size_t left = 0;
    size_t right = data.size();
    while (right - left > 1)
    {
        size_t middle = std::midpoint(left, right);
        if (data[middle] ≤ value)
        {
            left = middle;
        }
        else
        {
            right = middle;
        }
    }
    return left;
}
```

Формально данный отрывок кода называется *определением шаблона функции*. Выражение `T` в этом коде называют *аргументом шаблона*, первую строчку *заголовком шаблона*, а остальной код *телом шаблона*. Отметим, что вместо ключевого слова `class` в первой строчке возможно использование ключевого слова `typename`. Эти ключевые абсолютно эквивалентны в контексте аргументов шаблона и не имеют отношения к классам как к фиче языка C++. Продолжая аналогию с макросами, мы можем явно попросить компилятор создать функцию по написанному нами шаблону:

```
template size_t binarySearch<int>(
    std::span<int const> data, int value);
```


Понимать эту строчку следует как подстановку вместо неё тела шаблона с аргументами заменёнными на указанные в ней. В данном случае все вхождения имени `T` заменяется на `int`. Формально процесс подстановки аргументов в шаблон и получение настоящей функции языка C++ называется *инстанциацией*. Результат процесса инстанциации, то есть получаемая в результате функция, называется *специализацией*. Само выражение написанное выше называют *явной инстанциацией* (причина такого названия станет ясна ниже). Имя функции, сгенерированной этой явной инстанциацией, выглядит как `binarySearch<int>`, а вызов соответственно как `binarySearch<int>(data, 42)`.

Однако, явные инстанциации – весьма редко используемый на практике инструмент. В отличие от макросов, компилятор способен сам отслеживать с какими аргументами необходимо инстанцировать шаблон. Можно считать, что в недрах компилятора для каждого шаблона функции хранится таблица соответствий набора аргументов определению получаемой в результате функции. Таблица заполняется лениво по мере необходимости, то есть встретив в коде вызов функции `binarySearch<U>`, где `U` – какой-то конкретный тип, компилятор либо использует уже имеющееся определение функции из таблицы, либо предварительно сгенерирует новое и заполнит им ячейку в таблице. Этот процесс формально называется *неявной инстанциацией*, или просто *инстанциацией*.

Обратим внимание несколько важных деталей о взаимодействии шаблонов функций и структуры многофайловых проектов. Определение шаблона функции не является определением функции, а значит на него не распространяется *one definition rule*. Следовательно мы можем помещать определения шаблонов как и в заголовочных (`.hpp`) файлах, так и в компилируемых (`.cpp`) файлах. Однако есть веская причина почти всегда помещать шаблоны в заголовочных файлах: для инстанциации шаблона необходимо иметь полное тело шаблона в текущем файле. Из этого вытекает один из главных недостатков шаблонов. Так как каждая единица трансляции компилируется из соответствующего `.cpp` файла независимо¹, у компилятора нет выбора кроме как заново инстанцировать шаблон в каждой единице трансляции, заново компилировать полученную инстанциацию в

¹Это позволяет системам сборки запускать несколько процессов компиляции разных файлов параллельно на многоядерных системах

бинарный код и вкладывать его в каждый получаемый объектный файл. На практике это приводит к «раздуванию» размера итогового бинарного файла программы и увеличению времени компиляции. Как мы увидим дальше, первая из этих проблем уже не актуальна.

Узнав про неявную инстанциацию у читателя мог возникнуть закономерный вопрос: а зачем вообще в языке C++ есть механизм явной инстанциации? Перед тем как ответить на этот вопрос, нам понадобится познакомиться с новым понятием, *объявлением шаблона функции*:

```
template<class T>
size_t binarySearch(std::span<T const> data, T value);
```

Если инстанциация определения шаблона функции с конкретными аргументами приводит к определению функций, инстанциация объявления шаблона функции приводит к объявлению функций. Это значит, попытавшись инстанцировать шаблон функции имея только объявление шаблона, но не определение шаблона, объектный файл скомпилируется, но программа в целом не слinkуется, если в каком-то другом объектном файле не было инстанцировано определение. На практике возможность отдельно определять и объявлять шаблоны функций используется двумя способами.

Во-первых, традиционно каждая пара .cpp и .hpp файлов отвечает одному мини-модулю программы, и в заголовочном файле принято оставлять исключительно публичное API этого модуля. Это позволяет читающему код легко использовать имеющийся код не влезая в детали имплементации, а также не компилировать заново все использующие заголовочный файл модули при изменении деталей имплементации. Шаблоны функций же нарушают эту традицию, у нас нет выбора кроме как помещать имплементацию (определение) прямо в заголовочный файл, иначе использующий наш модуль код не сможет инстанцировать шаблон. С точки зрения механики компиляции с этим поделать ничего нельзя, однако с точки зрения синтаксиса и удобства чтения кода пользователем мы можем улучшить ситуацию использовав объявления шаблонов следующим образом:

```
// Файл binarySearch.hpp
#pragma once

#include <span>
#include <cstdint>
```

```
template<class T>
size_t binarySearch(std::span<T const> data, T value);

#include "binarySearch.ipp"

// Файл binarySearch.ipp

template<class T>
size_t binarySearch(std::span<T const> data, T value)
{
    ...
}
```

Мы разделили заголовочный файл на 2 половины: файл .hpp (header) с объявлениями шаблонов и файл .ipp (implementation) с определениями шаблонов, где первый подключает второй в свой конец. Таким образом пользователю чтобы использовать наш модуль достаточно прочитать содержимое файла .hpp, не обязательно пролистывать весь код чтобы узнать какие функции есть в модуле. На тривиальном примере из одной функции это звучит слегка абсурдно, но при написании крупных модулей с десятками функций эта техника сильно помогает облегчить жизнь читающего код.

Во-вторых, есть ситуации, когда шаблон предполагается использовать с очень ограниченным семейством типов, например только с примитивными числовыми типами. В этой ситуации мы можем пойти ещё дальше и полностью спрятать тело шаблона от пользователя в .cpp файл, инстанцировав явно его для необходимых нам аргументов.

```
// Файл binarySearch.hpp
#pragma once

#include <span>
#include <cstdint>

template<class T>
size_t binarySearch(std::span<T const> data, T value);

// Файл binarySearch.cpp

template<class T>
size_t binarySearch(std::span<T const> data, T value)
```

```
{  
    ...  
}  
  
template size_t binarySearch<int>(  
    std::span<int const> data, int value);  
template size_t binarySearch<float>(  
    std::span<float const> data, float value);
```

В случае с двоичным поиском вероятно такое разделение излишне, ведь пользователь может захотеть использовать его со своими типами данных, но время от времени эта техника действительно всплывает на практике. Именно для неё в язык C++ и были добавлены явные инстанцииции.

Упражнения

- Задумайтесь, что произойдёт, если попытаться инстанцировать шаблон `binarySearch` с аргументом типа `Person`, описанным ниже. Попробуйте написать это. Что нужно сделать, чтобы функция искала человека по возрасту? Легко ли было это понять?

```
struct Person  
{  
    std::string name;  
    uint32_t age;  
};
```

- Напишите шаблон функции `pairLess` от одного аргумента с типом произвольной специализации `std::pair`, возвращающую `true` если первый элемент меньше второго, а иначе `false`. Обратите внимание, типы первого и второго элемента пары могут отличаться. О синтаксисе шаблона с несколькими аргументами читателю предлагается догадаться самостоятельно.

1.2. constexpr, consteval, constexpr

Начиная с C++11 в языке по чуть-чуть начали появляться новые инструменты метапрограммирования, выступающие и альтернативой и дополнением к шаблонам: титулярные ключевые слова. В общем и целом

все они предназначены для переноса выполнения некоторых вычислений из рантайма в компайлтайм. Рассмотрим следующий пример.

```
enum class LogLevel
{
    Info, Warning, Error, Fatal
};

std::unordered_map<LogLevel, std::string> logLevelStrings
{
    {Color::Info, "INFO"},
    {Color::Warning, "WARNING"},
    ...
};

...

#define LOG_WARN(msg) \
std::printf("[%s] (%s:%d): %s",
    logLevelStrings.at(LogLevel::Warning),
    __FILE__, __LINE__,
    msg);
```

Подобный код для логирования можно встретить во многих промышленных проектах. Чем плох наш вариант? Обратить внимание стоит на поиск элемента в `std::unordered_map`. Он хоть и работает за $O(1)$, но всё равно не бесплатен. Почему мы платим за поиск известного нам во время компиляции элемента в хеш-таблице временем в рантайме? Конечно же можно полностью отказаться от использования таблицы и вписать нужную строку прямо в макрос, но это решение не расширяется на большее число различной инфраструктуры, связанной с логированием. Мы быстро начнём копировать эту строку в разных местах и в итоге придём к нарушению правила одного источника истины². Вместо этого попробуем заменить таблицу на функцию:

```
enum class LogLevel
{
    Info, Warning, Error, Fatal
};
```

²SPOT, single point of truth

```
const char* logLevelToString(LogLevel level)
{
    switch (level)
    {
        case LogLevel::Info: return "INFO";
        ...
    }
    return "?";
}
```

Это уже чуть лучше, прыжок в `switch` скорее всего будет дешевле вычисления хеш-функции. Однако при вызове нашей функции мы всё ещё будем нагружать процессор лишней работой: если мы знаем на момент компиляции аргумент `level`, то мы можем знать и результат функции. Именно эту проблему решает ключевое слово `constexpr`. Дописав его перед сигатурой функции мы **запретим** компилятору компилировать эту функцию в традиционном смысле этого слова, и разрешим лишь только вычислять её результат во время компиляции.