

Метапрограммирование на C++

4 сентября 2022 г.

Оглавление

1	Введение в шаблоны	3
1.1	Шаблоны функций	3
1.2	Шаблоны методов	16
1.3	Шаблоны глобальных переменных	17
1.4	constexpr, consteval, constexpr	19
1.5	Констрейнты и концепты	30
1.6	Шаблоны типов	30

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 предопределённым 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 в следующем разделе, первая из этих проблем уже не совсем 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 прочесть 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.

Рассмотрим ещё одну возможность шаблонов функций, *явные специализации*². Напомним, что специализацией называется функция результат инстанцииции шаблона. Предположим, что наша целевая платформа поддерживает специальную функцию `fastIntSearch`, способную очень быстро найти индекс целого числа в массиве размера меньше, скажем, 16 элементов. Хотелось бы, чтобы наша функция `binarySearch` использовала эту инструкцию как только область поиска стала достаточно маленькой, но конечно же использовать её можно исключительно для типа данных `int`. Как нам изменить код специализации `binarySearch<int>` не поменяв код любых других специализаций? Ровно эту функцию выполняют явные специализации:

```

template<>
size_t binarySearch<int>(std::span<int const> data, int value)
{
    size_t left = 0;
    size_t right = data.size();
    while (right - left > 16)
    {
        size_t middle = std::midpoint(left, right);
        if (data[middle] ≤ value)
        {
            left = middle;
        }
        else
        {

```

²Вообще говоря официальное название – *явная полная специализация*

```
        right = middle;
    }
}
return fastIntSearch(data.subspan(left, right - left), value);
}
```

Если в области видимости на момент использования `binarySearch<int>` содержится и основное определение шаблона, и эта явная специализация, то для вызова будет использована явная специализация. Стоит обратить внимание, что на момент объявления явной специализации шаблон функции уже должен быть объявлен. Также заметим, что явная специализация является обычным определением функции, а поэтому может быть разделено на объявление и определение, где первое следует помещать в заголовочный файл, а второе в .cpp-файл:

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

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

Если поместить определение явной специализации в заголовочный файл, необходимо пометить её как `inline`, иначе мы нарушим ODR³.

В заключение раздела упомянем о *выведении типов*. До сих пор мы явно указывали шаблонные аргументы для специализаций в момент вызова или специализации. Работая с более сложными шаблонами необходимость указывать эти аргументы быстро становится обременяющей, поэтому в C++ был добавлен достаточно сложный механизм автоматического вывода типа шаблонных аргументов. Детали работы этого механизма будут рассмотрены в одной из следующих глав, а сейчас лишь скажем, что в большом количестве случаев указывать шаблонные аргументы не обязательно:

³One definition rule

```
void foo()
{
    std::vector<int> data{ ... };
    int value = 42;
    // Тип T автоматически будет выведен как int
    // на основании типа аргумента value.
    binarySearch(data, value);
}

// Аналогично для явных специализаций
template<>
size_t binarySearch(std::span<int const> data, int value)
{
    ...
}
```

Упражнения

- Задумайтесь, что произойдёт, если попытаться инстанциировать 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.
- Что произойдёт, если забыть объявить явную специализацию в заголовочном файле?
- Рассмотрим следующий код

```
// Часть интерфейса библиотеки
struct GraphicsState
{
    bool enableWireframe;
    std::string noiseTextureName;
    std::optional<PipelineDescription> pipelineDescription;
    ...
};

// Часть имплементации библиотеки
struct InternalGraphicsState
{
    bool enableWireframe{false};
    TexturePtr noiseTexture;
    std::optional<PipelineDescription> pipelineDescription;
    ...
};

InternalGraphicsState covertState(
    const GraphicsState& state)
{
    InternalGraphicsState result;
    result.enableWireframe = state.enableWireframe;
    if (!noiseTextureName.empty())
    {
        result.noiseTexture =
            getTexture(state.noiseTextureName);
    }
    result.pipelineDescription = state.pipelineDescription;
    ...

    return result;
}

struct InternalGraphicsStateDelta
{
    // std::nullopt означает отсутствие изменения
    std::optional<bool> enableWireframe;
    std::optional<Texture> noiseTexture;
    std::optional<Pipeline> pipeline;
    ...
};
```

```

InternalGraphicsStateDelta calculateDelta(
    const InternalGraphicsState& before,
    const InternalGraphicsState& after)
{
    InternalGraphicsStateDelta result;
    if (before.enableWireframe
        ≠ after.enableWireframe)
    {
        result.enableWireframe
            = after.enableWireframe;
    }

    if (before.noiseTexture ≠
        after.noiseTexture)
    {
        result.noiseTexture
            = after.noiseTexture;
    }

    if (before.pipelineDescription
        ≠ after.pipelineDescription)
    {
        result.pipeline = createPipeline(
            after.pipelineDescription)
    }

    ...

    return result;
}

```

Используя шаблоны функций и возможно макросы, избавьте код от копиасты логики и сделайте добавление новых полей-состояний в эти структуры более безболезненным. Для поддержки ”особенных” полей воспользуйтесь явной специализацией шаблонов функций.

1.2. Шаблоны методов

Помимо ”свободных” функций в языке C++ поддерживаются связанные с классом функции, называемые *методами*. Методы также можно ге-

нерировать при помощи шаблонов:

```
class Any
{
public:
    template<class T>
    T* castTo()
    {
        return reinterpret_cast<T*>(data_);
    }
private:
    void* data_;
};
```

Определение шаблонов методов также можно отделять от объявления:

```
template<class T>
T* Any::castTo() { ... }
```

Со специализациями же есть одна тонкая деталь, согласно стандарту их необходимо помещать вне объявления класса:

```
// Объявление специализации
template<>
int* Any::castTo<int>();
// Определение специализации
template<>
int* Any::castTo<int>() { ... }
```

Но компилятор MSVC позволяет помещать их и внутри самого класса. Использовать эту возможность разумеется не стоит, так как это делает код платформозависимым.

1.3. Шаблоны глобальных переменных

Искушённый знанием языка C читатель знает, что в объектном файле скомпилированной программы бывает два вида символов: функции и переменные. Если C++ разрешает генерировать символы функций из шаблонов, было бы странно запрещать генерировать символы переменных. И действительно, язык поддерживает *шаблоны переменных*, однако за исключением упомянутого в следующем разделе частного случая, используются они сильно реже шаблонов функций.

Синтаксис шаблонов переменных абсолютно аналогичен синтаксису шаблонов функций, как и набор языковых средств связанных с ним.

```
// Объявление шаблона переменной
template<class T>
extern std::vector<T> globalStorage;

// Определение шаблона переменной
template<class T>
std::vector<T> globalStorage;

// Явная инстанциация
template std::vector<int> globalStorage<int>;

// Явная специализация
template<
NotStdVector<bool> globalStorage<bool>;
```

Как можно понять из примера, в чистом виде шаблоны глобальных переменных в основном используются в качестве параметризованного глобального хранилища в процедурном коде.

Ранее уже упоминалось, что шаблоны повторно инстанцируются в разных объектных файлах, в результате чего символ специализации может содержаться в нескольких объектных файлах. Для функций эта особенность не страшна, так как соответствующий специализации символ указывает на неизменяемый код функции. Для переменных же возникает опасение: если одной переменной соответствует несколько символов с разным хранилищем, то на самом деле мы получим не одну переменную, а несколько независимо меняющихся переменных. Но на самом деле опасение напрасно, в результате статической линковки и для функций и для переменных будет оставлен только один их экземпляр в итоговом бинарном файле. На типичных платформах достигается это за счёт механизма *слабых* символов. При компиляции обычных функций и переменных в единице трансляции генерируются сильные символы. Если при линковке сильный символ с одинаковым названием содержится в нескольких единицах трансляции, линковщик выдаст ошибку. В случае же функций и переменных, являющихся специализациями шаблонов, либо помеченных как `inline`, компилятор генерирует слабые символы. Встретив несколько слабых символов с одинаковым именем, линковщик выкинет все из них кроме одного. Таким

образом в итоговой программе будет содержаться лишь один экземпляр каждой функции и переменной.

Стоит отметить, что с точки зрения стандарта C++ никаких символов не существует. Линковка шаблонов описана куда более абстрактным языком, и частенько оказывается оторванным от практики. В крупных проектах итоговая программа не редко разбита на множество динамических библиотек. Например Unreal Engine разбит на множество dll-файлов для возможности hot-reload отдельных модулей без полной перезагрузки редактора, а в использующих GNU core utilities операционных системах большая часть функционала вынесена в so-библиотеки для экономии памяти и удобства обновления отдельных модулей системы. Отличие динамических библиотек от статических в том, что нахождение расположения символов происходит не в момент компиляции программы, а прямо в процессе работы, через обращение к линковщику операционной системы. Таким образом в момент компиляции символы из статических библиотек копируются в итоговый бинарный файл нашей программы, потенциально выкидывая дубликаты слабых символов, а символы из динамических библиотек лишь порождают "заглушки" помогающие программе загружать необходимые символы по мере нужды. К сожалению, при загрузке символов в рантайме не происходит дедупликации слабых символов, а поэтому одна и та же специализация шаблонной переменной из разных so-файлов будет на самом деле ссылаться на разную память, иначе говоря это будут две разные, независимые переменные. По этой причине стоит избегать шаблонов переменных при проектировании публичных интерфейсов библиотек.⁴

1.4. constexpr, consteval, constinit

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  
{
```

⁴Более подробно о механизме динамической линковки можно почитать по [ссылке](#).

```

    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 истины⁵. Вместо этого попробуем заменить таблицу на функцию:

```

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

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

```

⁵SPOT, single point of truth

```

    ...
}
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 `consteval`. Дописав его перед сигнатурой функции мы **запретим** компилятору компилировать эту функцию в традиционном смысле этого слова, и разрешим лишь только вычислять её результат *прямо во время компиляции*:

```

consteval const char* logLevelToString(LogLevel level) { ... }

// ...

int main(int argc, char** argv)
{
    int level = 0;
    std::cin >> level;
    // Ошибка компиляции: LogLevel{level}
    // не является константным выражением
    std::cout << logLevelToString(LogLevel{level});
    // ОК! Всё выражение после `<<` будет
    // вычислено в момент компиляции, в
    // итоговом бинарном файле будет лишь вызов
    // `operator<<(std::cout, "FATAL")`.
    std::cout << logLevelToString(LogLevel::Fatal);
}

```

В случае же если мы не хотим запрещать вычисление функции в рантайме, но хотим чтобы компилятор вычислял результат вызова функции во время компиляции когда это возможно, следует использовать ключевое слово `constexpr` вместо `consteval`. Подчеркнём это дважды: для функций `consteval` означает "всегда в компайлтайме", а `constexpr` означает "можно в компайлтайме". Также заметим, что ключевые слова `constexpr` и `consteval` автоматически помечают функции как `inline`, поэтому тела таких функций можно и нужно помещать в заголовочные файлы, ведь

иначе компилятор не сможет вычислить функцию в момент компиляции⁶.

Обратим внимание на понятие *константного выражения*, упомянутое в примере выше. Для наших целей достаточно интуитивного его понимания как *выражения, которое можно вычислить в момент компиляции*, более точное определение содержится в стандарте. Примерами константных выражений являются все литералы и вызовы `constexpr` и `constexpr` функций от них. Контрпримером же является переменная `int level` в примере выше, так как в неё можно поместить данные, известные только в рантайме. Однако константные переменные могут как быть так и не быть константными выражениями в зависимости от константности выражения которым они инициализированы.

Ключевое слово `constexpr` можно также использовать с переменными. К типу таких переменных автоматически добавляется ключевое слово `const`, а выражение которым инициализируется такая переменная может быть только константным. Таким образом значение `constexpr`-переменной всегда известно во время компиляции, то есть является константным выражением, и не может быть изменено в рантайме. Более того, компилятор по возможности будет избегать включения такой переменной как символа в скомпилированную программу.

```
constexpr size_t DATA_AMOUNT = 42;
...
// DATA_AMOUNT скорее всего не породит символа
std::array<int, DATA_AMOUNT> data;

...

// Обратите внимание, тип MESSAGE --
// const char* const, то есть константный
// указатель на константный char.
constexpr const char* MESSAGE = "Fatal error!";

...
// MESSAGE породит символ в разделе rodata,
// в котором будет сохранена соответствующая строка
std::cout << MESSAGE;
```

Перекладывая этот инструмент на наш изначальный пример, мы можем

⁶При вычислении функций во время компиляции код на C++ на самом деле интерпретируется.

изменить код процедуры `logLevelToString` сделав его data-driven вместо code-driven:

```
struct LogLevelInfo
{
    LogLevel level;
    const char* name;
};

constexpr std::array<LogLevelInfo>
{
    LogLevelInfo{LogLevel::Info, "INFO"},
    ...
};

constexpr const char* logLevelToString(LogLevel level)
{
    for (auto [l, name] : logLevelInfos)
    {
        if (l == level)
        {
            return name;
        }
    }
    return "UNKNOWN";
}
```

Отметим что использовать `std::unordered_map` вместо массива структур не получится, так как стандарт пока что не требует от методов контейнера `std::unordered_map` наличия пометки `constexpr`, хотя теоретически написать хеш-таблицу которую можно использовать в константных выражениях возможно.

Наконец освятим последнее титулярное ключевое слово, `constinit`. Оно является "урезанной" версией слова `constexpr` и применимо только в отношении глобальных переменных⁷, и требует чтобы инициализатор переменной был константным выражением, не добавляя при этом константности к типу, а также не делая саму переменную константным выражением.

В прошлом разделе мы отмечали, что шаблоны переменных используются реже шаблонов функций, за исключением одного частного случая.

⁷Если быть точнее, переменных со статическим или тред-локальным временем жизни

Этот частный случай – шаблоны `constexpr`-переменных. Основная их ценность в возможности моделировать различные отображения типов в значения. Например, мы можем задать отображение, возвращающее `true` для типов с плавающей точкой и `false` иначе:

```
template<class T>
constexpr bool isFloatingPoint = false;

template<>
constexpr bool isFloatingPoint<float> = true;

template<>
constexpr bool isFloatingPoint<double> = true;

template<>
constexpr bool isFloatingPoint<long double> = true;
```

Однако используя только лишь шаблоны `constexpr`-переменных сами по себе нам не удастся выразить большинство полезных отображений, для этого нам понадобятся инструменты из следующих разделов. Пока что же узнаем об ещё один инструмент языка C++ использующий ключевое слово `constexpr`.

```
template<class T>
T dotProduct(std::span<const T> first,
             std::span<const T> second)
{
    T result = T{0};
    assert(first.size() == second.size());
    for (size_t i = 0; i < first.size(); ++i)
        result += first[i] * second[i];
    return result;
}
```

Пример выше содержит наивную имплементацию скалярного произведения для двух векторов со скаляром произвольного типа `T`. Если для пользовательских сложных типов `T` такая имплементация и может быть оптимальной, для типов с плавающей точкой большинство платформ предоставляет набор специальных векторизированных операций, в том числе и скалярное произведение для векторов ограниченного размера. Неплохо было бы оптимизировать эту функцию для случаев чисел с плавающей точкой. Если наша платформа поддерживает векторизацию только для `float`,

мы можем воспользоваться уже известной нам явной специализацией шаблонов функций, но что если платформа предоставляет набор перегруженных векторизированных функций для всех трёх видов чисел с плавающей точкой? Хотелось бы просто разобрать два случая:

```
template<class T>
T dotProduct(std::span<const T> first,
             std::span<const T> second)
{
    T result = T{0};
    assert(first.size() == second.size());
    if (isFloatingPoint<T>)
    {
        // Имплементация с использованием векторизации
        for (size_t i = 0; i < first.size(); i += 4)
            result += builtinScalarProduct4(
                first.data() + i, second.data() + i);
    }
    else
    {
        ... // Старая имплементация
    }
    return result;
}
```

Однако такой код не скомпилируется если перегрузки `builtinScalarProduct4` не существует для `T`, то есть функция будет работать только для встроенных типов с плавающей точкой. Решить эту проблему нам поможет конструкция `if constexpr`:

```
template<class T>
T dotProduct(std::span<const T> first,
             std::span<const T> second)
{
    T result = T{0};
    assert(first.size() == second.size());
    if constexpr (isFloatingPoint<T>)
    {
        ...
    }
    else
    {
        ...
    }
}
```

```

    }
    return result;
}

```

Эта вариация обычного `if` заставляет компилятор вычислять условие в скобках в момент компиляции, и полностью выкидывать из рассмотрения ненужную ветку. Это и позволяет нам писать в неактивной ветке семантически некорректный код, а именно вызов несуществующей перегрузки `builtinScalarProduct4`. Очень важно понимать, что условие *всегда* будет вычисляться в момент компиляции. С 20й версии стандарта языка в стандартной библиотеке появилась функция `std::is_constant_evaluated`. Она возвращает `true` если была вычислена в момент компиляции и `false` если в рантайме:

```

// alwaysTrue всегда будет true, так как
// компилятор всегда старается инициализировать
// глобальные переменные в момент компиляции
bool alwaysTrue = std::is_constant_evaluated();

void foo()
{
    // alwaysFalse всегда false
    bool alwaysFalse = std::is_constant_evaluated();
    std::cout << alwaysFalse;
}

```

Добавлена эта функция в язык была с целью упростить реализацию `constexpr` версий различных функций. Как пример вернёмся к функции `dotProduct`. Платформозависимые функции вроде `builtinScalarProduct4` обычно не помечены как `constexpr`, так как реализованы ассемблерными вставками. Это означает что даже если мы пометим наш шаблон функции `dotProduct` как `constexpr`, вычислить его специализацию для `float` компилятору не удастся. С первого взгляда может показаться, что достаточно добавить в условие выбора реализации функции `std::is_constant_evaluated` чтобы при вычислении `dotProduct` в компайлтайме всегда использовалась вторая реализация:

```

template<class T>
constexpr T dotProduct(std::span<const T> first,
    std::span<const T> second)

```

```

{
    T result = T{0};
    assert(first.size() == second.size());
    // Неправильно!
    if constexpr (isFloatingPoint<T>
        && !std::is_constant_evaluated())
    {
        ...
    }
    else
    {
        ...
    }
    return result;
}

```

Однако как мы уже подчёркивали, условие в конструкции `if constexpr` *всегда* вычисляется в момент компиляции, а поэтому добавление `&& !std::is_constant_evaluated()` никак не поменяло поведения функции, выражение `std::is_constant_evaluated()` всегда будет равно `true` в этом контексте. Правильным решением проблемы будет использование обычной конструкции `if`:

```

template<class T>
constexpr T dotProduct(std::span<const T> first,
    std::span<const T> second)
{
    T result = T{0};
    assert(first.size() == second.size());
    if (!std::is_constant_evaluated())
    {
        if constexpr (isFloatingPoint<T>)
        {
            // Имплементация через builtinScalarProduct4
            ...
        }
        else
        {
            // Наивная имплементация
            ...
        }
    }
    else
    {

```

```

        // Наивная имплементация
        ...
    }
    return result;
}

```

Однако использование `std::is_constant_evaluated` с обычным `if` все ещё не удовлетворительно для некоторых ситуаций. Рассмотрим следующий пример:

```

constexpr int fooImpl1(int) { ... }

int fooImpl2(int) { ... }

constexpr int foo(int i)
{
    if (std::is_constant_evaluated())
    {
        // Ошибка компиляции!
        return fooImpl1(i);
    }
    else
    {
        return fooImpl2(i);
    }
}

```

В данном отрывке кода вызов функции `fooImpl1` вызовет ошибку компиляции в силу того, что это выражение не возможно вычислить на этапе компиляции, так как не известно значение `i`, а как мы помним, `constexpr` функции запрещено вычислять в рантайме. Естественной реакцией на такую ошибку было бы дописать `constexpr` после `if`, но как мы только что видели, такой подход ошибочен. Для решения этой проблемы в 23й версии языка была добавлена новая конструкция, `if constexpr`:

```

constexpr int foo(int i)
{
    if constexpr
    {
        return fooImpl1(i);
    }
    else

```

```
{  
    return fooImpl2(i);  
}
```

Её семантика аналогична `if (std::is_constant_evaluated())`, но при этом ненужная ветка выкидывается как и в случае с `if constexpr`, что предотвращает ошибку компиляции.

В заключение раздела поговорим об ограничениях `constexpr` функций и переменных. На момент добавления этих возможностей в язык ограничений было очень много: нельзя было выделять память на куче, использовать циклы и условия, и многое другое. Однако с каждой следующей версией стандарта ограничений становится всё меньше, поэтому при написании `constexpr`-функций не стоит намеренно ограничивать себя в использовании языковых инструментов, а для сложных случаев всегда можно посмотреть точные требования в [справочнике](#). Есть лишь одно исключение, достойное явного упоминания: в константных выражениях запрещено иметь `undefined behavior`. Более того, компилятор обязан следить за этим фактом и выдавать ошибку компиляции если в процессе вычисления константного выражения возникла ситуация обозначенная стандартом как `undefined behavior`. Например, если в ходе вычисления произошло переполнение знакового целого типа, компилятор обязан об этом сообщить, а программа считается некорректной.

Упражнения

- Придумайте как добиться семантики `if consteval` используя инструменты C++20.
- Возьмите произвольный модуль написанного вами кода и попробуйте заставить его функционировать в компайлтайме используя материал данного раздела. Не забывайте заглядывать на `srppreference` встречающие ограничения `constexpr` функций.

1.5. Констрейнты и концепты

1.6. Шаблоны типов