Метапрограммирование на C++

Санду Р.А.

13 августа 2023 г.

Оглавление

Глава 1

Введение в шаблоны

Основное применение шаблонов берёт свои корни в необходимости обобщать алгоритмы и структуры данных на произвольные типы, а как было замечено ещё в 70е годы, «алгоритмы + структуры данных = программы», [1]. 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 же 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 новые 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. Когда коллега решит оптимизировать этот алгоритм посредством линейного поиска на массивах, влезающих в кэш-линию, он будет крайне недоволен необходимостью вписать эту оптимизацию в 5, а то и 10 мест. 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 такой подход на самом деле не решает основную проблему: алгоритм не был обобщён, вместо этого было сделано три разных алгоритма с разными названиями. Это не позволит использовать его в других обобщённых алгоритмах, не продолжая при этом 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 использовани-

ем макросов. Но опасность злоупотребления макросами 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. Возможная реализация обобщения алгоритма `binarySearch` с использованием макросов приведена ниже.

```
#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 отличии от C, позволяет перегружать функции. Это решает проблему суффиксов, но не избавляет нас от необходимости использовать макросы.

Альтернативным подходом от мира C был бы отказ от использования

конкретных типов, заменяя их на указатели на байты и используя указатель 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. Но 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(data[0]), 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 получится.

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 больше нет нужды заниматься низкоуровневыми манипуляциями байтами. Однако компромисс состоит в производительности: мы больше не можем передавать на вход алгоритму обычный массив данных, необходимо передавать массив указателей на данные. Также вызов виртуального метода-компаратора ещё дороже, чем вызов указателя на функцию-компаратор.

Итак, перечислим недостатки, имеющиеся в рассмотренных подходах:

— сложный пользовательский интерфейс,
— дублирование логики,
— игнорирование синтаксиса языка (макросы),
— протекание абстракций (необходимость работать с байтами),
— необходимость в индерекции данных (интерфейсы),
— необходимость в индерекции фукнций (компараторы).

К счастью, язык C++ предоставляет нам инструмент, позволяющий избежать каждой из этих проблем: шаблоны. Как можно догадаться из объёма данного пособия, компромисс шаблонов состоит в сложности их использования. Больше всего они похожи на первый из рассмотренных подходов. Шаблон — аналог макроса, учитывающий синтаксис языка ещё до подстановки аргументов. Язык поддерживает возможность написания шаблонов лишь для ограниченного числа сущностей в коде: функций, переменных, классов, псевдонимов и концептов.[1] Обобщение алгоритма `binarySearch` при помощи шаблонов выглядит следующим образом.

```
template<class T>
```

[1] Являются ли концепты строго говоря шаблонами — спорный вопрос.

```
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*, строку 1 *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 возможности 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` заменятся 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 — не часто используемый на практике инструмент. В отличии от макросов, компилятор способен самостоятельно 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.[2] Этот процесс формально называется *неявной инстанциацией*, или просто *инстанциацией*.

Обратим внимание на 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 независимо,[3] у компилятора нет выбора кроме как заново инстанцировать шаблон в каждой единице трансляции, заново компилировать полученную инстанциацию в 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

[2] На самом деле этот механизм несколько сложнее в силу разделения общей компиляции программы на компиляцию объектных файлов и их линковку.

[3] Это позволяет системам сборки запускать несколько процессов компиляции разных файлов параллельно на многоядерных системах.

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 генерации определения функции, инстанциация объявления шаблона функции приводит к генерации объявления функции. Это значит, попытавшись инстанциировать шаблон функции имея только объявление шаблона, но не определение шаблона, объектный файл скомпилируется, но программа в целом не слинкуется, 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 программиста нет выбора, кроме как помещать имплементации (определения шаблонов) в заголовочный файл, иначе использующий этот мини-модуль код не сможет инстанцировать шаблон и получить определение функции. С точки зрения механики компиляции, с этим поделать ничего нельзя, однако, с точки зрения синтаксиса и удобства чтения кода пользователем, ситуацию можно улучшить использовав объявления шаблонов следующим образом.

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

#include <span>
#include <cstddef>
```

```
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)
{
   ...
}
```

Заголовочный файл был разделён 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, человеку, желающему использовать этот модуль и не интересующемуся внутренним его устройством, достаточно прочитать содержимое файла `.hpp`, а открывать и пролистывать содержимое файла `.ipp` нужды нет, аналогично обычному, не шаблонному коду. В нашем примере преимущество не велико, однако для заголовочных файлов с десятками шаблонов эта техника незаменима.

Во-вторых, есть ситуации, когда шаблон предполагается использовать с очень ограниченным семейством типов, например, только с примитивными числовыми типами. В этой ситуации возможно полное разделение объявления и определения. В заголовочном файле будет помещено лишь объявление шаблона, а определение будет скрыто от пользователя в `.cpp` файле, где также при помощи явных инстанциаций будут сгенерированы все необходимые специализации. Нахождением же соответствий между специализациями объявления и специализациями определения будет линковщик.

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

```
#include <span>
#include <cstddef>

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)
{
  ...
}

// Специализации binarySearch для int и float
// будут помещены в текущей единице трансляции
// и будут видны линковщику извне.
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 с произвольными типами данных, но время от времени эта техника действительно используется на практике. Именно ради неё в язык C++ и был добавлен механизм явных инстанциаций.

Рассмотрим ещё одну возможность шаблонов функций, *явные специализации*.[4] Напомним, что специализацией называется функция результат инстанциации шаблона. Предположим, что целевая платформа поддерживает специальную функцию `fastIntSearch`, способную очень быстро найти индекс целого числа в массиве размера меньше, скажем, 16 элементов. Хотелось бы, чтобы функция `binarySearch` использовала эту инструкцию как только область поиска стала достаточно маленькой, но исключительно

[4] Вообще говоря официальное название — *явная полная специализация*.

для типа данных `int`. Как изменить код специализации `binarySearch<int>`, не поменяв при этом код всех остальных специализаций? Ровно это позволяет сделать следующий отрывок кода.

```
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
    {
      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 этот отрывок кода, называемый *явной специализацией*, то использована будет именно эта, явно имплементированная отдельно от шаблона 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 ODR.[5]

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 выведения 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:

```
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)
{
```

[5] One definition rule.

```
  ...
}
```

Упражнения

1) 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;
};
```

2) 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.
3) Что произойдёт, если забыть объявить явную специализацию в заголовочном файле?
4) Рассмотрим следующий код.

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

// Часть имплементации библиотеки
struct InternalGraphicsState
{
```

```
  bool enableWireframe;
  TexturePtr noiseTexture;
  std::optional<PipelineDescription> pipelineDescription;
  ...
};

InternalGraphicsState convertState(
  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 копипасты 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 генерировать 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 тонка деталь, согласно стандарту их необходимо помещать вне объявления класса:

```
// Объявление явной специализации
template<>
int * Any::castTo<int>();
template<>
// Определение явной специализации
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++ позволяет 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 параметризированного 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 оторванной от практики. В крупных проектах итоговая программа не редко разбита на множество динамических библиотек. Например, Unreal Engine разбит на множество `dll`-файлов для возможности горячей перезагрузки[6] отдельных модулей без полной перезагрузки редактора, а в использующих GNU core utilities операционных системах бóльшая 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` или `dll` файлов будет на самом деле ссылаться на разную память, иначе говоря, это

[6] Hot reload, возможность перекомпилировать часть проекта и подгрузить изменения в уже работающий процесс без его перезагрузки.

будут две разные, независимые переменные. По этой причине стоит избегать шаблонов переменных при проектировании публичных интерфейсов библиотек.[7]

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
{
  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

[7] Более подробно о механизме динамической линковки можно почитать в [2].

элемента в `std::unordered_map`. Он хоть и работает за $O(1)$, но всё равно не бесплатен. Пользователю приходится платить за поиск известного 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. Достаточно скоро эта строка начнёт копироваться в различные места, что в приведёт к нарушению правила одного источника истины.[8] Хранение строк в константах тоже не будет являться лучшим решением, так как может быть потеряна строгость типизации в различных функциях системы, а именно элементы перечисления будут заменены на `const char *`. Вместо этого заменим таблицу на функцию:

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

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

Прыжок в `switch` однозначно будет дешевле вычисления хеш-функции, однако работа по выполнению этого прыжка всё ещё лишняя: если аргумент `level` известен на момент компиляции, то теоретически должно быть известно и возвращаемое значение функции. Именно эту проблему решает ключевое слово `consteval`. Дописав его перед сигнатурой функции, компилятору будет **запрещено** компилировать эту функцию в традиционном

[8] SSOT, single source of truth [3].

смысле этого слова, и разрешено лишь 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});
  // OK! 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 запрещать вычисление функции в рантайме нет необходимости, но желательно желательно иметь возможность вычислять результат вызова функции во время компиляции когда это возможно или необходимо, 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 компиляции.[9]

Обратим внимание на понятие *константного выражения*, упомянутое в примере выше. Для наших целей достаточно интуитивного его понима-

[9] При вычислении функций во время компиляции код на C++ на самом деле интерпретируется, и происходит это именно во время компиляции единицы трансляции, а не во время линковки.

ния как *выражения, которое можно вычислить в момент компиляции*, более точное определение содержится в стандарте. Примерами константных выражений являются все литералы и вызовы `constexpr` и `consteval` 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 неявно 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 изменить код процедуры `logLevelToString` сделав его data-driven вместо code-driven:

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

constexpr std::array logLevelInfos
  {
    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 переменных,[10] и требует, чтобы инициализатор переменной был константным выражением, не добавляя при этом константности к типу, а также не делая саму переменную константным выражением. **1.пример использования**

[10] Если быть точнее, переменных со статическим или тред-локальным временем жизни.

В прошлом разделе мы отмечали, что шаблоны переменных используются реже шаблонов функций, за исключением одного частного случая. Этот частный случай — шаблоны `constexpr`-переменных. Основная их ценность состоит 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 не удастся выразить большинство полезных отображений, для этого потребуется инструменты из следующих разделов.

Пока что же осветим 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 пользо-

вательских 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 целевая платформа поддерживает векторизацию только для `float`, можно воспользоваться уже изученной 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
    // (баг допущен намерено)
    for (size_t i = 0; i < first.size(); i += 4)
      result += builtinScalarProduct4(
        first.data() + i, second.data() + i);
  }
  else
  {
    ... // Старая имплементация
  }
  return result;
}
```

Но 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 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, что позволяет писать в неактивной ветке семантически некорректный код, а именно вызов несуществующей перегрузки `builtin_ScalarProduct4`. Очень важно понимать, что условие *всегда* будет вычисляться в момент компиляции и всегда должно быть константным выражением. 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 пометить 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;
}
```

Но, как уже было подчёркнуто, условие в конструкции if constexpr *всегда* вычисляется в момент компиляции, а поэтому добавление && !std::is_constant_evaluated()) полностью выключило первую ветку условия, ведь 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:

```
consteval 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 `consteval` функции запрещено вычислять в рантайме. Естественной реакцией на такую ошибку было бы дописать `constexpr` после `if`, но такой подход ошибочен, как только что было продемонстрировано. Для решения этой проблемы в 23й версии языка была добавлена новая конструкция, `if consteval`:

```
constexpr int foo(int i)
{
  if consteval
  {
    return fooImpl1(i);
  }
  else
  {
    return fooImpl2(i);
  }
}
```

Её семантика аналогична `if (std::is_constant_evaluated())`, но при этом ненужная ветка выкидывается компилятором из рассмотрения, 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. Но 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

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

1.5. Ограничения и концепты

Ознакомляясь с разелом 1.1, пытливый читатель мог заметить аспект, в котором обобщение кода при помощи шаблонов уступает объектно-ориентированному подходу. Любой обобщённый алгоритм или контейнер накладывает некоторые ограничения на типы данных, с которыми он может работать. Обобщив алгоритм двоичного поиска при помощи ООП, мы получили явный список требований в виде интерфейа `IComparable`, накладываемый на типы данных, с которыми пользователь захочет запустить этот алгоритм. В случае же шаблона функции `binarySearch`, пользователю не

предоставляется явных требований на типы, с которыми можно инстанцировать этот шаблон. Однако требования эти существуют: инстанциация провалится, если выражение `data[middle] ⩽ value` не корректно, иначе говоря, если не найдётся подходящей перегрузки `operator⩽`. В случае двоичного поиска это ограничение на типы можно посчитать очевидным, однако даже в случае с присовением конкретного функционального объекта к `std::function` требования уже не очевидны.

До 20го стандарта C++ ограничения и требования на типы, используемые в шаблонах, приходилось писать в комментариях и документации, либо угадывать по печально известным своей сложностью ошибкам инстанциации. В 20м же стандарта появился набор языковых возможностей, позволяющих явно задать ограничения на типы, используемые в шаблонах, и даже объеденить несколько требований в именованный предикат, в некотором аналогичный интерфейсам ООП. Рассмотрим следующий отрывок кода.

```
template<class T>
constexpr bool is_comparable1 = requires { T{} ⩽ T{}; };
```

Мы объявили шаблон `constexpr`-переменной и инициализировали его новой для читателя конструкцией, `requires`-*выражением*. Подобные выражения можно использовать только внутри шаблонных сущностей, они всегда вычисляются на этапе компиляции, а результат их вычисления — булево значение, `true` если все строчки написанные в фигурных скобках после `requires` успешно скомпилировались бы, если встретились в обычном коде, и `false` иначе. Стоит обратить внимание на то, что код, написанный внутри `requires`-выражения, фактически никогда не будет запущен или вычислен, ни в рантайме, ни в компайлтайме. Подчеркнём ещё раз, что комплиятор лишь «попробует скомпилировать» этот код, но ни в коем случае не будет пытаться его запускать. Строчек в фигурных скобках может быть несколько, как в следующем примере.

```
template<class T>
constexpr bool is_comparable2 = requires
  {
    T{} < T{};
    T{} > T{};
```

```
    T{} ⩽ T{};
    T{} ⩾ T{};
  };
```

Заметим, что `requires`-выражения в предыдущих двух примерах будут иметь значение `false`, если тип `T` нельзя инициализировать без аргументов, так как мы использовали внутри `requires`-выражений конструкцию `T{}`. Если подобное поведение нежелательно, можно воспользоваться параметрами `requires`-выражений.

```
template<class T>
constexpr bool is_comparable3 = requires(T a, T b) { a ⩽ b; };
```

В данном отрывке мы просим компилятор проверить, скомпилируется ли отрывок кода `a ⩽ b`, в предположении что у нас уже откуда-то есть два разных значения типа `T`. Интересной деталью стандарта являетс тот факт, что если `requries`-выражение имеет значение `false` для любого шаблонного, то программа считается некорректной, при чём от компилятора не требуется диагностировать эту ошибку, как в следующем примере.

```
template<class T>
constexpr bool invalid = requires
  {
    // не валидно для любого T, программа не корректна!
    new int[-(int)sizeof(T)];
  };
```

Выражения, перечисленные через точку запятой в фигурных скобках, формально называют *требованиями*. Встречавшиеся нам до сих пор требования называют *простыми*. Они представляют из себя произвольные выражения язка C++, например, вызовы функций или операторов. Существует ещё несколько типов требований, поддерживаемых `requires`-выражениями, а именно, *типовые*, *составные* и *вложенные*. Однако их рассмотрение придётся на некоторое время отложить.

Рассмотренные нами `requires`-выражения позволяют спросить компилятор о том, скомпилируется ли какой-либо отрывок кода, но может быть не очевидно, как они помогут решить проблему явных требований к типам

в шаблонах. Для решения этой проблемы нам необходимо ознакомится с ещё одним нововведением C++20, ограничениями. Начнём с немного надуманного примера.

```
template<typename T>
  requires (sizeof(T) < 128)
void foo(T t);
```

Синтаксис второй строчки называется *requires*-предложением[11] и семантически задаёт *ограничение* на тип, с которым можно инстанцировать шаблон функции `foo`. Таким образом, при попытке интстанцировать этот шаблон с типом `T`, размер которого превышает или равен 128 байтам, компилятор выдаст ошибку подстановки. Синтаксически, после ключевого слова `requires` можно написать любое булево константное выражение. Например, используя определённый выше шаблон `constexpr`-переемменной `is_comparable3`, мы наконец-то можем написать близкую к идеальной сигнатуру для шаблона `binarySearch`.

```
template<class T>
  requires is_comparable3<T>
size_t binarySearch(std::span<T const> data, T value);
```

Теперь при попытке использовать этот шаблон с типом, для которого не определён `operator`$\leqslant$, компилятор выдаст более понятную ошибку компиляции, а человек, читающий данный код, сразу увидит, что функцию можно использовать лишь с типами, которые можно сравнивать. Заметим, что `requires`-предложение также можно писать после сигнатуры функции, а также в качестве ограничения можно использовать `requires`-выражения. Комибинруя эти две возможности, корректен следующий код.

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

Здесь вместо использования параметров `requires`-выражений в качестве переменной для формирования выражения сравнения был использован параметр самой функции. Заметим, что такая конструкция корректна только

[11] От английского «requires clause»

при написании `requires`-предложений после сигнатуры, так как в ином случае аргументы функции ещё не считаются определёнными.

2.Про сами собственно концепты и short-circuit булевых операторов

1.6. Шаблоны типов

1.7. SFINAE

3.Тут про старый способ жить без концептов, но что важнее, про immediate context и вот это всё

1.8. Разрешение имён

4.Обязательно про dependent base class name resolution

Глава 2

Введение в препроцессор

Эта глава проклята, не читайте её.

Глава 3

Шаблоны и дизайн

Как и любую другую языковую возможность, важно не только понимать семантику шаблонов, но и уметь выстраивать с их помощью хороший дизайн модулей приложения. Данная глава посвящена различным устоявшимся техникам и подходам к дизайну с использованием шаблонов. [4]

3.1. Трейты

3.2. Curiously recurring template pattern

3.3. Policy based design

3.4. Mixins

3.5. Tag dispatch

3.6. Полиморфизм

5.Общаяя информация про полиморфизм, классификация его видов, етц

3.6.1. Стирание типов

3.6.2. Открытые мультиметоды

3.6.3. Точки кастомизации

6.Ниблоиды, `tag_invoke`

3.7. Генерация иерархий

Глава 4

Избранные этюды

Эта глава содержит набор избранных трюков, хаков и этюдов мира метапрограммирования. Автор не несёт ответственности за судьбу читателя в случае попытки применить описанные в этой главе вещи в продакшене.

Список литературы

1. *Wirth N.* Algorithms + Data Structures = Programs. — Prentice-Hall, 1976. — ISBN 978-0-13-022418-7.
2. *Wienand I.* PLT and GOT - the key to code sharing and dynamic libraries. — 2011. — URL: https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html.
3. Single source of truth. — URL: https://en.wikipedia.org/wiki/Single_source_of_truth.
4. *Alexandrescu A.* Modern C++ Design: Generic Programming and Design Patterns Applied. — USA : Addison-Wesley Longman Publishing Co., Inc., 2001. — ISBN 0201704315.