Лекция 11 — Шаблоны

Содержание

1. Общие представления о шаблонах
    1. Мотивация
    2. Шаблонные параметры и аргументы
    3. Идентификаторы шаблонов
    4. Шаблоны функций
    5. Шаблоны классов
    6. Шаблоны переменных
    7. Шаблоны как члены класса
    8. Шаблоны псевдонимов
2. Инстанцирование шаблонов
3. Зависимые имена
4. Выведение шаблонных аргументов
5. Специализация шаблонов
    1. Мотивация
    2. Полная специализация
    3. Частичная специализация
6. Шаблоны и перегрузка
7. SFINAE
8. Константы времени компиляции
9. Снова выведение типов
10. Вариадические шаблоны

”В C++ мы не говорим «пропущена звёздочка». Мы говорим «error C2664: ’void std::vector<block, std::allocator<_Y>>::push_back(const block&)’: cannot convert argument 1 from ’std::_Vector_iterator<std::_Vector_val<std::_Simple_types <block>>>’ to ’block&&’», *и это прекрасно.”*

— Конфукций, 239 г. до н.э.

Общие представления о шаблонах

Мотивация

Можно сказать, что основное преимущество процедурного программирования заключается в соблюдении принципа Don't Repeat Yourself. Это вообще замечательный принцип, который позволяет писать понятный и, что главное, легко редактируемый код. Например, если вынести логику взятия максимума из двух целочисленных значений в отдельную функцию:

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

то исправление или изменение этой логики сведётся к исправлению или изменению функции вместо потенциальной редактуры каждого вхождения конструкции `a > b ? a : b` в код программы — это не говоря уже о том, что запись с использованием тернарного оператора сильно менее наглядна, чем запись `max(a, b)`. Для читателя всё это не должно быть открытием.

Если же нам требуется поддерживать эту логику для данных различных типов — скажем, `int`, `float` и `std::string`, — то мы (особенно если представить, что мы в 1983 году) неизбежно вынуждены столкнуться с определенными проблемами. На данный момент нам известно два способа решить эту задачу.

Первый — перегрузить функцию `max` для всех нужных типов:

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

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

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

Очевидно, чем плох этот метод. Он порождает множество одинаковых функций, отличающихся только сигнатурой — более того, в перспективе это множество бесконечно, и для каждого нового типа мы обязаны самостоятельно доопределять новую функцию. Все преимущества, связанные с редактированием кода, при таком подходе мгновенно теряются.

Альтернативный вариант заключается в использовании текстовых макросов:

```
#define max(a, b) a > b ? a : b
```

Это решает проблему множественного копирования, но приносит с десяток новых, связанных, в частности, с отладкой и типобезопасностью. Проблемы макросов подробно обсуждались ранее, поэтому сейчас лишь заключим, что и это решение нам не подходит.

Таким образом мы приходим к неутешительному выводу о том, что мы, в общем-то, не знаем, как нормально работать с параметрически полимфорными объектами, то есть использовать идентичные механизмы обработки данных различных типов. Из этих же соображений в язык C++ была введена концепция **шаблонов**.

Шаблон позволяет единственным объявлением опрелелить *семейство* объектов, объединённых одной логикой. Он представляет собой «образец» функции, класса или переменной, который сам по себе не привязан к конкретным типам параметров. Важно, что сам он не является никаким из перечисленных объектов и никак в памяти не представлен.

Как уже было отмечено, шаблоны являются механизмом реализации параметрического полиморфизма, причем *статического*, поскольку сопоставление типов в этом случае происходит на этапе компиляции (см. *Инстанцирование шаблонов*). Они также реализуют парадигму *обобщённого программирования* и служат основой для техники *метапрограммирования*, которая будет подробнее рассмотрена отдельно.

Шаблон объявляется согласно следующему синтаксису:

Ключевое слово `templat e`	$\Longrightarrow$	Список параметров в угловых скобках	$\Longrightarrow$	Объявление

Прежде, чем сформировать полноценное представление о явлении шаблонов, формализуем некоторые связанные с ним понятия.

Шаблонные параметры и аргументы

Шаблонные *параметры* — т.е. значения, указанные при объявлении шаблона — и *аргументы* — т.е. значения, передаваемые при его использовании — могут быть представлены как:

- имя типа;
- константа;

- шаблон;
- пакет из типов, констант и шаблонов (см. *Вариадические шаблоны*)

При обращении к идентификатору шаблона имеет место явление *подстановки* (*substitution*), при котором компилятор просто подставляет переданный аргумент везде, где встречалось имя соответствующего параметра. Совершенно незатруднительно придумать пример, в котором это приведет к ошибке.

> 1. **Типовые параметры и аргументы**
>
> Типовые шаблонные параметры задаются при помощи ключевых слов `typedef` и `class`:
>
> ```
> template<typename T>
> // template<class T>
> ```
>
> Практически во всех контекстах, связанных с шаблонами, `typename` и `class` эквивалентны. Особые случаи будут рассмотрены далее.
>
> В теле шаблона `T` является `typedef`-декларацией, связанной с именем типа, который, собственно, и передаётся как типовый аргумент:
>
> ```
> template<int>
> ```

> 2. **Константные параметры и аргументы**
>
> При описании константных шаблонных параметров необходимо явно указывать их тип. Допускаются следующие типы константных параметров:
>
> - lvalue-ссылка;
> - целочисленный тип;
> - вещественнозначный тип;
> - указатель;
> - enum;
> - `std::nullptr_t`
> - `auto` (см. *Ключевое слово* `auto`)
>
> ```
> template<int x>
> ```
>
> При описании константного параметра допускается объявление массива или функции, однако они всё равно автоматически конвертируются в соответствующие указатели.

Константные шаблонные аргументы могут быть представлены любым инициализатором или явно заданным константным выражением.[1] Если тип шаблонного аргумента нельзя привести к типу шаблонного параметра, компилятор выдаст ошибку.

В случае, когда шаблонный аргумент можно интерпретировать и как имя типа, и как выражение, он всегда интерпретируется как имя типа:

```
template<typename T>
void f() {
    std::cout << "template<typename T>" << std::endl;
}

template<int T>
void f() {
    std::cout << "template<int T>" << std::endl;
}

int main() {
     f<int()>();
}
```

3. **Шаблонные параметры и аргументы**

В качестве шаблонного параметра может также быть описан другой шаблон, формируя *шаблонный шаблонный параметр*. Шаблонным шаблонным аргументом здесь может выступать только шаблон класса:

```
template<typename T>
class templated_class;

template<typename T>
void templated_function() {}

template<typename T1, template<typename> typename T2>
void foo() {}

int main() {
     foo<int, templated_class>; // OK!
     foo<int, tempalted_function>; // CE!
}

// прим. Этот пример несколько забегает вперед, но иначе, увы, никак
```

Имя параметра не может быть переопределено в теле шаблона, но при определенных условиях может затемняться, ровно как и затемнять другие имена. [2]

Всем шаблонным параметрам, за исключением пакетных, могут быть назначены значения по умолчанию согласно примерно тем же принципам, что и в случае функций: [3]

```
template<typename T, int val1 = 5, float val2 = 31.04>
```

Шаблоны в том числе поддерживают опережающие объявления, что позволяет доопределять (но не переопределять) списки аргументов по умолчанию.

Идентификаторы шаблонов

Запись вида `template-name<template-arg-list>` называется *идентификатором* шаблона и однозначно определяет объект, который он описывает. Два одинаковых шаблонных идентификатора задают один и тот же объект.

Идентификаторы считаются одинаковыми, если:

- они относятся к одному шаблону;
- их типовые шаблонные аргументы совпадают;
- их шаблонные шаблонные аргументы совпадают;
- все их шаблонные параметры, определенные по шаблонным аргументам, эквивалентны. [4]

Критерии валидности идентификатора шаблона приблизительно такие же, как и в случае вызова функции. В общих чертах они сводятся к следующему набору условий:

- аргументов не больше, чем параметров (либо последним параметром является пакет);
- каждому параметру без значения по умолчанию сопоставлен аргумент;
- тип каждого аргумента успешно приводится к типу соответствующего ему параметра.

```
template<class T, T::type n = 0>
class X;

struct S
{
    using type = int;
};

using T1 = X<S, int, int>; // ошибка: слишком много аргументов
using T2 = X<>;            // ошибка: первый параметр не имеет аргумента по
```

```
                          //          умолчанию
using T3 = X<1>;          // ошибка: значение 1 не соответствует первому
                          //          параметру
using T4 = X<int>;        // ошибка подстановки во второй параметр
using T5 = X<S>;          // OK!
```

Шаблоны функций

Проблема, с которой началось обсуждение шаблонов, подразумевала работу с семейством параметрически полиморфных функций. Собственно, именно они позволяют наиболее удачным образом эту работу организовать.

Шаблон функции объявляется согласно всем тем же правилам, что и обычные функции:

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

Практически, эта запись говорит компилятору: «после объявления шаблона описан „чертёж“, подстановкой в который вместо `T` конкретного типа данных получается полноценная функция». Чтобы дать ему указание создать конкретную *шаблонную функцию*, достаточно указать в коде идентификатор шаблона:

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

int main() {
    std::cout << max<int>(238, 30); // 238
}
```

Процесс инициализации шаблонных функций будет более подробно рассмотрен далее (*см. Инстанцирование шаблонов*).

Шаблоны классов

Так же, как и в случае с функциями, мы можем использовать шаблоны для объявления семейств классов:

```
template<typename T>
class SmallArray {
```

```
  public:
    T[10] arr;
};
```

Практически все механизмы и термины, касающиеся шаблонов классов, аналогичны шаблонам функций. Различия в технической реализации будут указаны далее.

Шаблоны переменных

Начиная с C++14 в языке существует возможность создавать также шаблоны переменных:

```
template<typename T>
const T pi = T(3.14159263104);
```

Хотя такая опция и может показаться странной, это бывает полезно для, например, математических констант или, в некоторых случаях, для общей параметризации логики программы.

Шаблоны как члены класса

Объявления шаблонов могут встречаться в теле класса:

```
class Collection {
  public:
    template<typename T>
    void AddObject(T* obj_ptr) {
         add_((void*)obj_ptr);
    }
}
```

Шаблоны методов не могут быть виртуальными или перегружать родительские виртуальные методы, а также опеределять конструкторы копирования или деструкторы.

Шаблонные переменные внутри класса представляют статические поля.

Шаблоны псевдонимов

C++ позволяет, помимо прочего, объявлять шаблонизированные сокращения для имён типов:

```
template<typename T>
using Vec = std::vector<T>
```

```
int main() {
    Vec<int> my_vector; // эквивалентно std::vector<int>
}
```

Инстанцирование шаблонов

Как было отмечено, шаблоны по себе не являются ни функциями, ни классами, ни переменными, ни типами — они вообще никак не представлены в коде. В случае, например, с шаблонами функций, в этом легко убедиться, посмотрев на таблицу символов:

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

int main() {
}

/* Таблица символов:
0000000000000000 T main */
```

Если же указать в `main` идентификатор шаблона, в таблице символов появится запись о шаблонной функции с соответствующим именем:

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

int main() {
    max<int>(238, 3104);
}

/* Таблица символов:
0000000000000000 T main
0000000000000000 W int max<int>(int const&, int const&) */
```

ⓘ Определение

> Процесс генерации полноценной сущности по её шаблону называется его **инстанцированием** (*template instantiation*).

Процесс, описанный выше, называется *неявным инстанцированием*, поскольку мы лишь обращаемся к функции, которая должна существовать, передавая всю остальную работу компилятору. Если при этом функции не существует, он её генерирует, иначе — использует уже созданную. Для неявного инстанцирования определение шаблона должно быть доступно в каждом файле, где оно используется.

Инстанцировать шаблон можно явным объявлением в глобальном поле видимости:

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

template int max<int>(const int& a, const int& b);
// тот же эффект, что и в примере выше
```

Если при явном инстанцировании указать ключевое слово `extern`, шаблонная сущность будет объявлена, но не будет определена. Это запретит неявное инстанцирование, и компилятор будет обязан обратиться к явному определению, находящемуся где-либо ещё. Если этого определения нет, мы, очевидно, сталкиваемся с ошибкой линковки. Указание `extern` позволяет компилятору не инстанцировать один и тот же шаблон несколько раз в разных единицах трансляции, что в определенной степени «облегчает» программу.

Важный нюанс состоит в том, что неявное инстанцирование «лениво», поскольку компилятор генерирует только те объекты, к которым мы обращаемся, что особенно заметно при работе с шаблонными классами:

```
template<typename T>
class Bug {
  public:
    T bad() {
        T x;
        return x + 1;
    }
};

int main() {
```

```
    Bug<std::string> obj;
}
```

Здесь метод `std::string bad()`, очевидно, не должен работать, поскольку мы не можем прибавить к строке число. Ожидаемо, что не вызовет ошибку компиляции на этапе определения шаблона, поскольку компилятору заведомо неизвестно, с какими типами он должен работать (это отдельная проблема, которая ещё будет поднята далее). Менее ожидаемо, что это не вызовет ошибки и на этапе определения шаблонного класса, поскольку метод `bad()` после этого всё ещё не генерируется. Программа упадёт лишь если мы попытаемся обратиться к некорректному методу напрямую.

Стоит отметить, что если сделать метод `bad()` виртуальным:

```
template<typename T>
class Bug {
  public:
    virtual T bad() {
        T x;
        return x + 1;
    }
};

int main() {
    Bug<std::string> obj;
}
```

то программа также упадёт, хотя обращения к нему всё так же нет. Это происходит по уже известным нам причинам: компилятору нужно создать виртуальную таблицу и поместить туда указатель на этот метод, а для этого его необходимо определить.

В свою очередь, явное инстанцирование обязывает компилятор определить сразу все поля и методы класса, поэтому следующий код:

```
template<typename T>
class Bug {
  public:
    T bad() {
        T x;
        return x + 1;
    }
};

template class Bug<std::string>;
```

уже приведёт к ошибке.

Эти соображения закрепелены в понятии двухфазного поиска (*two-phase name lookup*) — такое название в Стандарте носит процесс инстанцирования шаблонов. Как ни странно, он проходит в две фазы:

1. в момент определения шаблона он проходит синтаксические проверки;
2. в момент инстанцирования шаблонный аргумент подставляется вместо шаблонного параметра.

Некоторые компиляторы не полностью поддерживают первую фазу, откладывая всю происходящую в ней работу до момента инстанцирования.

Следует помнить, что некоторые неочевидные ошибки в шаблонах (в частности, шаблонах классов) вы можете обнаружить лишь в самый последний момент, поэтому стоит очень внимательно и осознанно подходить к их проектированию.

Зависимые имена

В теле шаблона функции или класса значение некоторых имен может изменяться от одной инстанции к другой — в частности, типы и выражения могут зависеть от шаблонных параметров:

```
template<typename T, int val>
void foo() {
     T(val) obj; // obj зависит от T и val
     std::cout << obj->Method();  // obj->Method() зависит от T
}
```

Соответствующие имена называются **зависимыми**. Связывание и поиск производятся по-разному для зависимых и независимых имён.

Независимые имена связываются в момент определения шаблона. Это связывание сохраняется, даже если в момент инстанцирования существует более подходящий кандидат:

```
void g(float x) {
     std::cout << "void g(float x)\n";
}

template<typename T>
```

```
void f() {
     g(1);
}

void g(int x) {
     std::cout << "void g(int x)\n";
}

int main() {
     g(1); // void g(int x)
     f<int>(); // void g(float x)
}
```

Если при этом значение независимого имени изменилось к моменту инстанцирования (например, если изначально оно имело неполный тип), мы столкнемся с ошибкой компиляции:

```
class C;

template<typename T>
void f() {
     C x;
}

class C {
    int x;
};

int main() {
     f<int>();
}
```

Связывание зависимых имён откладывается до стадии поиска. Поиск зависимых имён откладывается до момента, когда становятся известны шаблонные аргументы. Поиск при этом проверяет лишь объявления, которые видны в момент определения шаблона, а не в момент инстанцирования. Иными словами, объявление новой функции после объявления шаблона не сделает её видимой для этого шаблона:

```
namespace noo {
    template<typename T>
    void WriteObject(const T& obj) {
        std::cout << "Value = " << obj << std::endl;
    }
}
```

```
std::ostream& operator<<(std::ostream& out, const std::vector<int>& vec) {
    std::cout << "This is a vector!";
    return out;
}

int main() {
    std::vector<int> vec;
    noo::WriteObject(vec); // ошибка: нет доступа к оператору <<
}
```

Это, впрочем, не распространяется на ADL:

```
namespace noo {
    template<typename T>
    void WriteObject(const T& obj) {
        std::cout << "Value = " << obj << std::endl;
    }
}

class MyClass {};

std::ostream& operator<<(std::ostream& out, const MyClass& obj) {
    std::cout << "Object of my class";
    return out;
}

int main() {
    MyClass obj;
    noo::WriteObject<MyClass>(obj); // OK!
}
```

Это ограничение исключает неочевидные нарушения ODR, которые могли бы возникнуть, например, при работе с заголовочными файлами.

Некоторые имена считаются *относящимися к текущей инстанции*, что позволяет обнаруживать некоторые ошибки уже на стадии определения шаблона — например, к этой категории относится имя шаблонного класса, имя шаблона с параметрами, вложенные классы и проч. [5]

Для разрешения неоднозначностей, связанных с зависимыми именами, используются ключевые слова `typename` и `template`, которые обозначают компилятору, что соответствующее имя гарантированно является либо именем типа, либо именем шаблона соответственно:

```
template<typename T1>
class MyClass {
  public:
    template<typename T2>
    void foo() {
        std::cout << "MyClass foo()\n";
    }
};

int p = 1;

template<typename T>
void boo() {
    MyClass<T> obj;
    obj.foo<int>(); // '<' парсится как 'меньше'
    obj.template foo<int>(); // OK!
    std::vector<T>::const_iterator* p; // '*' парсится как 'умножить'
    std::vector<T>::typename const_iterator* p; // OK!
}

int main() {
    boo<int>();
}
```

Ключевое слово `typename` может использоваться только после квалификатора `::`, тогда как `template` имеет место после `.`, `->` и `::`.

Выведение шаблонных аргументов

Для инстанцирования шаблона функции необходимо, чтобы каждый шаблонный параметр был известен, однако не каждый шаблонный аргумент обязательно указывать явно. В ряде случаев компилятор сам определит тип параметра исходя из переданных в шаблонную функцию аргументов — это называется *Template argument deduction*:

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

int main() {
```

```
    std::cout << max(238, 3104); // эквивалентно max<int>(238, 3104)
}
```

Механизм автоматического определения типов при инстанцировании помимо прочего позволяет использовать шаблонные операторы, поскольку в инфиксной форме явное указание шаблонных аргументов невозможно.

Прежде, чем перейти непосредственно к выведению типов, компилятор преобразует аргументы: для всех нессылочных типов он переводит массивы и функции в соответствующие указатели, а также игнорирует константность верхнего уровня.

Выведение типов имеет место, когда соответствующий аргументу параметр соответствует одной из 35 возможных форм, которые могут быть вложены друг в друга и обрабатываться рекурсивно. Среди этих форм — типы, ссылки, массивы, инициализаторы и идентификаторы шаблонов. Все из них не будут приведены в силу малой полезности данной информации. [6]

Тип шаблонных параметров *не* выводится в случае, если:

1. параметр указан в спецификаторе имени слева от `::`;
2. параметр является частью арифметического выражения;
3. параметр является частью аргумента по умолчанию;
4. параметр является функцией с неопределённой перегрузкой, и проч. [7]

Если тип не был определен для данного конкретного параметра, компилятор использует тип, который сопоставил этому параметру где-либо в другом месте. Если и этого не произошло, мы сталкиваемся с ошибкой компиляции:

```
template<typename T>
struct Identity {
    typedef T type;
};

template<typename T>
void foo(typename Identity<T>::type value) {};

int main() {
    foo(3); // CE
}
```

Если из разных аргументов тип одного и того же параметра `T` определился несколькими различными способами, это также приведет к ошибке компиляции:

```
template<typename T>
void foo(T x, T y);

int main() {
    foo(1, 2.3);
}
```

Специализация шаблонов

Мотивация

В некоторых случаях бывает полезно делать исключения из общей логики поведения шаблона, не нарушая при этом синтаксической целостности. В случае с шаблонами функций (и особенно если мы доверяем выведение типов параметров компилятору) этого можно достичь путём объявления отдельной нешаблонной функции, которой будет перенаправляться вызов с конкретными аргументами:

```
template<typename T>
void foo(T x) {
     std::cout << "Template function\n";
}

void foo(int x) {
     std::cout << "Global function\n";
}

int main() {
    foo(238); // Global function
    foo(31.04); // Template function
}
```

Нужный результат достигается благодаря тому, что в рамках разрешения перегрузки нешаблонная функция всегда более предпочтительна, чем шаблонная, если они подходят одинаково хорошо. (*подробнее см. Шаблоны и перегрузка*)

С шаблонами классов, впрочем, провернуть такое уже не выйдет, поскольку никакого аналога перегрузки для них не существует. Сама задача для них всё так же актуальна: известным примером здесь служит `vector<bool>`, на хранение элементов которого не хотелось бы выделять по 8 бит, из которых лишь один хранит информацию. Для него имеет смысл реализовать несколько отличную логику работы с элементами, чтобы минимизировать перерасход памяти, при этом семантически это должен быть всё тот же

вектор с прежним пользовательским интерфейсом. В этой ситуации нам на помощь приходит *специализация шаблонов* — механизм, позволяющий задать отдельное поведение для конкретного набора шаблонных аргументов.

Полная специализация

Специализация поддерживается практически всеми видами шаблонов, хотя, как только что было отмечено, наиболее актуальна она именно для шаблонов классов.

Специализация задаётся после основного определения шаблона. Чтобы это сделать, необходимо указать конструкцию `template<>`, после чего объявить объект с тем идентификатором, поведение которого необходимо изменить:

```
template<typename T>
void foo() {
    std::cout << "Common template\n";
}

template<>
void foo<int>() {
    std::cout << "Specialized template\n";
}

int main() {
    foo<bool>(); // Common template
    foo<int>(); // Specialized template
}
```

Специализации необходимо объявлять прежде, чем произойдёт самое первое неявное инстанцирование.

В случае с шаблонами функций в сигнатуре специализации можно опускать шаблонные параметры, если компилятор может их вывести:

```
template<typename T>
void foo(T x);

template<>
void foo(int x); // эквивалентно void foo<int>(int x)
```

Частичная специализация

Порой бывает необходимо создавать вариации шаблонов не для одного набора аргуметов, а для целых семейств — например, если мы хотим реализовать отдельную

логику для указателей на произвольный тип или для случая, когда типы шаблонных аргументов совпадают. В C++ этого позволяет добиться механизм *частичной специализации*. Она задаётся аналогично полной специализации, за исключением двух аспектов:

1. в конструкции `template<>` обязательно указываются все используемые шаблонные параметры;
2. список аргументов в идентификаторе шаблона нельзя опустить.

```
template<typename T, typename U>
class A {};

template<typename T>
class A<T, T> {}; // частичная специализация для случая T == U
```

К огромному счастью, частичная специализация не поддерживается шаблонами функций, поскольку это сделало бы поиск имён настолько неадекватно сложным процессом, что даже самые терпеливые приверженцы языка перестали бы на нём писать.

Частичная специализация обязана, как ни странно, что-либо специализировать, то есть список шаблонных аргументов не должен повторять список параметров в основном объявлении. Синтаксис частичной специализации также не может использоваться для объявления полных специализаций. Оба этих случая приводят к ошибке компиляции.

Шаблоны и перегрузка

Так же, как и обычные функции, шаблоны функций поддерживают перегрузку:

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

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

Непосредственно механизм разрешения перегрузок уже обсуждался ранее, поэтому сейчас мы коснёмся лишь ключевых моментов, относящихся непосредственно к

шаблонам.

Так, нешаблонные функции всегда отличаются от шаблонных и, как уже было отмечено, всегда имеют более высокий приоритет при разрешении перегрузок. Две шаблонные функции с одинаковым возвращаемым значением и списком параметров отличны друг от друга и могут различаться списком явно переданных шаблонных аргументов:

```
template<int i>
void foo() {
    std::cout << "template<int i> void foo()\n";
}

template<int i, int j>
void foo() {
    std::cout << "template<int i, int j> void foo()\n";
}

int main() {
     foo<238>();
    foo<31, 04>();
}
```

Два шаблона функции эквивалентны, если: [8]

1. они объявлены в одной и той же области видимости;
2. их имена совпадают;
3. их списки параметров имеют одну длину, и при этом сами параметры эквивалентны;
4. все встречающиеся в них выражения с участием шаблонных параметров эквивалентны.

Для компилятора эквивалентность означает, что эти шаблоны являются описанием одного и того же объекта, и их можно воспринимать как одну и ту же запись.

Два выражения с участием шаблонных параметров *функционально эквивалентны*, если они не эквивалентны, но при любом наборе аргументов результат их вычисления совпадает.
Два шаблона функции *функционально эквивалентны*, если они эквивалентны с поправкой на возможную функциональную эквивалентность некоторых выражений с участием шаблонных параметров. Согласно Стандарту, присутствие в программе функционально эквивалентных, но не эквивалентных шаблонов делает код технически некорректным, хотя не обязывает компилятор выдавать ошибку.

В случае, когда при вызове шаблонной функции возникает неоднозначность, имеет место *частичное упорядочение* (*partial ordering*) шаблонов, в результате которого компилятор выбирает «наиболее специализированную» перегрузку. Неформально, шаблон `A` более специализирован, чем шаблон `B`, если он принимает более узкое подмножество типов, чем `B` — на практике это означает, что `B` может принять аргументы `A`, но не наоборот.

Важно, что в разрешении перегрузок участвуют только нешаблонные функции и перегрузки основного шаблона, но не специализации — лишь когда шаблон функции уже был выбран наиболее подходящим, компилятор рассматривает его специализации. В некоторых ситуациях такое поведение может показаться несколько неинтуитивным:

```
template<typename T>
void foo(T x) {
    std::cout << "Primary template\n";
}

template<typename T>
void foo(T* x) {
    std::cout << "Template overload\n";
}

template<>
void foo<int*>(int* x) {
    std::cout << "Primary template specialization\n";
}

int main() {
    int* ptr = nullptr;
    foo(ptr); // Template overload
}
```

Поэтому стоит запомнить, что **специализации — это не перегрузки**. Во избежание неопределенностей и неожиданностей рекомендуется при работе с шаблонами функций использовать именно перегрузки или даже нешаблонные функции.

SFINAE

«Ошибка подстановки — это не ошибка» (*Substitution Failure Is Not An Error*) — так звучит принцип, согласно которому происходит разрешение перегрузки шаблонов функций. Это значит, что если подстановка шаблонного аргумента *в объявлении перегрузки* (важно: ошибки в теле функции SFINAE не прощает!) не увенчалась успехом, то есть если

в результате получилось синтаксически некорректное выражение, компилятор проигнорирует текущую перегрузку без сообщения об ошибке:

```
template<typename T>
void foo(T x, typename T::id) {}; // у int нет поля id!

template<typename T>
void foo(T x) {};

int main() {
     foo<int>(238); // OK!
}
```

Помимо того, что SFINAE исторически служит средством безопасности, которое позволяет сохранять целостность программы при добавлении посторонних шаблонов, оно также позволяет намеренно скрывать некоторые перегрузки шаблонов функций от компилятора — именно в этом и состоит его прелесть.

SFINAE — крайне мощный инструмент, которому можно найти десятки различных применений и который практически служит основой метапрограммирования в C++. Он очень тесно связан с `type_traits`, концептами и ограничениями, но это совсем другая история, к которой мы вернемся в отдельном разговоре о метапрограммировании, поскольку эта тема несколько выходит за рамки обсуждения шаблонов.

Константы времени компиляции

Засчёт того, что значение константных шаблонных аргументов по определению не зависит от этапа выполнения программы, компилятор почти всегда может оптимизировать связанные с ними вычисления. Несколько поразмыслив, мы можем попытаться эксплуатировать эту особенность:

```
template<int i>
int Factorial() {
    return Factorial<i - 1>() * i;
}

template<>
int Factorial<1>() {
    return 1;
}
```

```
int main() {
    std::cout << Factorial<5>();
}
```

Хотя, скорее всего, без дополнительных инструкций компилятор честно развернёт все рекурсивные вызовы, при указании достаточно сильных оптимизаций он действительно просто выполнит алгоритм и выведет число, не генерируя никакого кода.

Ключевое слово `constexpr`, введённое в C++11 и значительно дополненное в последующих стандартах, позволяет явно указать, что вычисления выражений, в которых встречается переменная или функция, *можно* производить во время компиляции.
В случае с переменными указание `constexpr` имеет также эффект ключевого слова `const`; в случае с функциями — ключевого слова `inline`, причём если функция была единожды помечена `constexpr`, то любое ее дальнейшее объявление также должно быть помечено `constexpr`.

Переменная может быть объявлена как `constexpr`, если:

1. она определена в момент объявления;
2. она имеет литеральный тип;
3. она инициализируется константным выражением;

Функция может быть объявлена как `constexpr`, если:

1. аргументы и возвращаемое значение имеют литеральные типы;
2. если это конструктор или деструктор, соответствующий класс не имеет виртуальных баз;
3. тело функции представлено в форме `= default` или `= delete` или не содержит ничего из следующего:
   - определения переменных нелитеральных типов;
   - определения статических или потоковых переменных;
   - ключевое слово `goto`.

Если все аргументы `constexpr`-функции являются константами времени компиляции, её значение также вычисляется на этапе компиляции; в противном случае она работает как обычная фукнция. Именно в этом заключается их основное удобство.

Мы также можем создавать классы, состояние которых вычисляется на этапе компиляции — для этого они должны иметь `constexpr` конструкторы и деструкторы. Для таких классов имеются следующие дополнительные ограничения:

- они не могут иметь виртуальных баз;
- все вызываемые конструкторы должны также быть `constexpr`-функциями;

- все подобъекты обязаны иметь `constexpr`-деструкторы.

Начиная с C++17, `constexpr` также может указываться в условном операторе сразу после `if`, чтобы позволить его значению вычисляться в момент компиляции. В отличие от обычного условного оператора, значение которого вычисляется в момент выполнения программы, ложная ветвь `constexpr if` выражения отбрасывается сразу после вычисления предиката, который, очевидно, также должен поддерживать вычисление в момент компиляции. Такие выражения чаще всего имеют место при работе со сложными шаблонными конструкциями, — обычно метафункциями, — и поэтому подробно рассмотрены не будут.

В C++20 были также введены ключевые слова `consteval`, которое гарантирует, что функция *обязана* выполниться при компиляции, и `constinit`, которое гаранитрует, что переменная *обязана* быть инициализирована при компиляции, но позволяет изменять её в момент выполнения программы.

Снова выведение типов

Компилятору можно поручить выведение типов и за пределами шаблонных параметров, используя ключевое слово `auto`. Оно может заменять собой идентификатор типа в объявлении переменной, и тогда её тип будет автоматически выведен согласно описанным ранее правилам:

```
auto x = 5; // int x = 5;
auto y = {1, 2, 3}; // std::initializer_list x = {1, 2, 3};
const auto z = 31.04; // const double z = 31.04;
auto w = x; // double w = x;
```

Его также можно указывать в списках параметров функций или как их возвращаемое значение — в первом случае такая запись заменяет собой шаблон (в Стандарте это называется *Abbreviated function template*):

```
void foo1(auto x) {}; // template<typename T> void foo1(T x);
```

Для выведения типов также используется ключевое слово `decltype`. Условно говоря, оно несет ту же семантику, что и `auto`, но сохраняет ссылки и константность.
В частности, конструкцию `auto` можно заменить на `decltype(auto)`:

```
const int x = 5;
```

```
decltype(auto) y = x; // const int y = x;
```

Ключевое слово `auto` также можно использовать в конструкциях, называемых *структурными связками* (*Structured bindings*), позволяя присваивать значения сразу группе переменных:

```
int a[3] = {1, 2, 3};

auto [x, y, z] = a; // x = 1, y = 2, z = 3
```

В этом случае выведение типа происходит согласно механизмам `decltype`.

`auto` и `decltype(auto)` также можно использовать для итерации по коллекциям:

```
int arr[4] = {3, 1, 0, 4};
for (auto& x : arr) {
     // ...
}
```

и, более того, при работе с `auto` прямо внутри этой итерации можно применять структурные связки:

```
std::map<int, float> mapa = {{238, 31.04}, {31, 0.4}};

for (auto& [key, value] : mapa) {
     // ...
}
```

Хотя они могут показаться крайне удобными и мощными инструментами, `auto` и `decltype` не следует использовать без меры — во многом из-за их неочевидного поведения, ибо мы уже знаем, что алгоритм выведения типов не самый простой и, откровенно говоря, не самый надежный. Этот блок плохо оставлять без пугалки, поэтому приведём один маленький безобидный пример, позаимствованный с просторов Хабра:

```
auto data = 134.29f;
auto v = data;
int dpcount = 0;
static double EPS = 1e-3;
while(v -(int)v > EPS){
     v *= 10;
     ++dpcount;
```

```
    std::cout<<"data:" << data << " " << v - (int)v << std::endl;
}
```

Достаточно стереть `f` в первой строке, и цикл никогда не кончится. Странно, что для корректной работы программы мы должны уточнять оператору *автоматического* выведения типа, какой конкретно тип надо вывести, правда?

Подытожим цитатой людей значительно более мудрых, чем автор или читатель:

> **Google C++ Code Style**
>
> Используйте выведение типов только для повышения читаемости или безопасности кода, и не используйте его ради избежания неудобств, связанных с явным указанием конкретного типа. Оценивая, стал ли код понятнее, помните, что ваши читатели не обязательно являются членами вашей команды или знакомы с вашим проектом. Поэтому типы, которые вам кажутся лишним шумом, зачастую будут предоставлять полезную информацию другим людям.

Вариадические шаблоны

Так же, как и обычные функции, шаблоны функций поддерживают переменное число аргументов, переданных через эллипсис:

```
template<typename... Types>
void foo() {};

int main() {
    foo<int>();
    foo<int, std::string, void*>();
    foo<>();
}
```

Переданные шаблонные параметры можно раскрыть, указав идентификатор, который за них отвечает, вместе с эллипсисом:

```
template<typename... Types>
void foo(Types...) {};

int main() {
    foo<int>(238);
```

```
    foo<int, std::string>(30, "orange");
    foo<>();
}
```

Более того, эллипсис в контексте раскрытия относится именно к выражению, после которого непосредственно следует — это позволяет, например, применять некоторый оператор в отдельности к каждому из переданных шаблонных аргументов:

```
template<typename T>
T inc(T x) {
    std::cout << x << ' ';
    return x + 1;
}

template<typename... Types>
void foo(Types...) {};

template<int... Values>
void inc_foo() {
    foo(inc(Values)...);
}

int main() {
    inc_foo<1, 2, 3>(); // 1 2 3
}
```

Если мы хотим непосредственно получить доступ к каждому из аргументов, нам придется прибегнуть к рекурсии:

```
template<typename... Args> // общий шаблон
void Print(Args...);

template<typename Current, typename... Args> // рекурсивная специализация
void Print(Current cur, Args... args) {
    std::cout << cur << ' '; // выводим текущий элемент
    Print<Args...>(args...); // вызываем шаблон от оставшихся
}

template<> // точка завершения рекурсии
void Print<>() {};

int main() {
    Print(1, 31.04, "string");
}
```

Вариадические шаблоны используются для функций-оберток и шаблонов коллекций произвольного размера (в частности, кортежей):

```
// пример простейшей реализации класса Tuple

template<typename... Args>
struct Tuple;

template<typename Head, typename... Tail>
struct Tuple<Head, Tail...> : Tuple <Tail...> {

    typedef Head value_type;
    typedef Tuple<Tail...> base_type;
    Head head;
    base_type& base = static_cast<base_type&>(*this);

    Tuple(Head h, Tail... tail)
        : Tuple<Tail...>(tail...)
        , head(h)
    {};

};

template<>
struct Tuple<> {};
```

Источники

1. А. П. Хвастунов — Лекции по основам программирования на C++, 1 семестр, 2025
2. Владимир Семенякин — Просто о шаблонах C++ (https://habr.com/ru/articles/599801/)
3. cppreference.com — Templates (https://en.cppreference.com/w/cpp/language/templates.html)
4. cppreference.com — constexpr specifier (https://en.cppreference.com/w/cpp/language/constexpr.html)
5. cppreference.com — Placeholder type specifiers (https://en.cppreference.com/w/cpp/language/auto.html)
6. cppreference.com — Structured bindings (https://en.cppreference.com/w/cpp/language/structured_binding.html)
7. @tri_tuza_v_karmane — C++ — Стреляем по ногам по-современному (https://habr.com/ru/articles/754662/)

1. см. раздел Manifestly constant-evaluated expressions (https://en.cppreference.com/w/cpp/language/constant_expression.html) ↩
2. см. раздел Name resolution for template identifiers (https://en.cppreference.com/w/cpp/language/template_parameters.html) ↩
3. см. раздел Default template arguments (https://en.cppreference.com/w/cpp/language/template_parameters.html) ↩
4. см. раздел Template argument equivalence (https://en.cppreference.com/w/cpp/language/template_arguments.html) ↩
5. см. раздел Current instantiation (https://en.cppreference.com/w/cpp/language/dependent_name.html) ↩
6. см. раздел Deduction from a type (https://en.cppreference.com/w/cpp/language/template_argument_deduction.html) ↩
7. см. раздел Non-deduced contexts (https://en.cppreference.com/w/cpp/language/template_argument_deduction.html) ↩
8. подробнее об эквивалентности параметров шаблонов и выражений с шаблонными аргументами см. в разделе Function template overloading (https://en.cppreference.com/w/cpp/language/function_template.html) ↩