Инстанцирование шаблона

Инстанцирование шаблона – это генерация кода функции или класса по шаблону для конкретных параметров.

```
template <class T>
bool lessThen7(T value) { return value < 7; }
```

```
lessThen7<int>(5); //  Инстанцирование
// bool print<int>(int value) {}

lessThen7<double>(5.0); //  Инстанцирование
// bool print<double>(double value) {}
```

Константы как аргументы шаблона

```
template <class T, size_t Size>
class Array
{
    T data_[Size];
};
```

```
Array<int, 5> a;
```

Ограничения на параметры шаблона не являющиеся типами

Так можно:

```
template <int N>
int foo()
{
    return N * 2;
}
```

А double нельзя:

```
template <double N> // Ошибка
void foo()
{
}
```

float тоже нельзя.

> Причины исторические, почему не исправлено до сих пор не знаю.

Параметры шаблона должны быть известны на этапе компиляции.

```
template <int N>
void foo() { }

int x = 3;
foo<x>(); // Ошибка
```

Константы на литералы можно:

```
template <int N>
void foo() {}
```

```
const int x = 3;
foo<x>(); // Ok
```

А с обычной константой нельзя:

```
int bar() { return 0; }

template <int N>
void foo() { }

const int x = bar();
foo<x>(); // Ошибка
```

Но если вычислять значение во время компиляции, то можно:

```
constexpr int bar() { return 0; }

template <int N>
void foo() {}

const int x = bar();
foo<x>(); // Ok
```

> constexpr говорит компилятору, что надо стараться вычислить значение на этапе компиляции

Нельзя использовать объекты класса:

```
struct A {};

template <A a> // Ошибка
void foo()
{
}
```

Можно указатель на const char:

```
template <const char* s>
void foo()
{
}
```

И это даже можно инстанцировать nullptr или 0:

```
foo<nullptr>();
foo<0>();
```

Но нельзя литералом:

```
foo<"some text">(); // Ошибка
```

Параметры шаблона по умолчанию

```
template <class X, class Y = int>
void foo()
{
}

foo<char>();
```

```
template <class T, class ContainerT = std::vector<T>>
class Queue
{
    ContainerT data_;
};
```

```
Queue<int> queue;
```

Специализация шаблона

```
template <class T>
class Vector
{
    ...
}

template <>
class Vector<bool>
{
    ...
};
```

Суммирование последовательности от n, до 0:

```
#include <iostream>

template <int n>
int sum();

template <>
int sum<0>() { return 0; }

// template <>
// int sum<1>() { return 1; }
// template <>
// int sum<2>() { return 2 + 1; }
// ...

template <int n>
int sum()
{
    return n + sum<n - 1>();
}

int main()
{
    std::cout << sum<3>() << '\n';
    return 0;
}
```

```
int sum<0>():
        mov     eax, 0
        ret
main:
        call    int sum<3>()
        call    operator<<(int)
        ret
int sum<3>():
        call    int sum<2>()
        add     eax, 3
        ret
int sum<2>():
        call    int sum<1>()
        add     eax, 2
        ret
int sum<1>():
        call    int sum<0>()
        add     eax, 1
        ret
```

Разбухание кода

Необдуманное использование шаблонов может привести к разбуханию кода, кода становится много, он перестает помещаться в кеш, что ведет к существенным издержкам.

Еще реализация суммирования

```
template <int n>
struct sum;

template <>
struct sum<0>
{
    static constexpr int value = 0;
};

template <int n>
struct sum
{
    static constexpr int value = n + sum<n - 1>::value;
};

int main()
{
    std::cout << sum<3>::value << '\n';
    return 0;
}
```

```
main:
        mov     esi, 6
        call    operator<<(int)
        ret
```

Вычисления времени компиляции

А стоит ли?

1. Сложно для понимания и поддержки
2. Замедляет компиляцию

> Вероятно лучшей альтернативой будет скрипт делающий вычисления и генерирующий C++ код из констант с рассчитаными значениями.

Псевдонимы типов

Старый способ:

```
typedef int Seconds;
typedef Queue<int> IntegerQueue;

Seconds i = 5;
IntegerQueue j;
```

Новый (рекомендуемый) способ:

```
using Seconds = int;
using IntegerQueue = Queue<int>;

Seconds i = 5;
IntegerQueue j;
```

Псевдонимы типов для шаблонов:

```
template <class T>
using MyQueue = Queue<T, std::deque<T>>;

MyQueue<int> y;
```

Новый синтаксис функций

```
auto foo() -> void
{
}
```

auto

Позволяет статически определить тип по типу выражения.

```
auto i = 5;
auto j = foo();
```

range-based for и auto

```
for (auto i : { 1, 2, 3 })
    std::cout << i;
```

```
for (auto& i : data)
    i.foo();
```

decltype

Позволяет статически определить тип по типу выражения.

```
int foo() { return 0; }

decltype(foo()) x = 5;
// decltype(foo()) -> int
// int x = 5;
```

```
void foo(decltype(bar()) i)
{
}
```

Определение типа аргументов шаблона функций

```
template <typename T>
T min(T x, T y)
{
    return x < y ? x : y;
}

min(1, 2); // ok
min(0.5, 2); // error
min<double>(0.5, 2); // ok
```

```
template <typename X, typename Y>
X min(X x, Y y)
{
    return x < y ? x : y;
}

min(1.5, 2); // ok
min(1, 0.5); // ok?
```

```
template <typename X, typename Y>
auto min(X x, Y y) -> decltype(x + y)
{
  return x < y ? x : y;
}

min(1.5, 2); // ok
min(1, 0.5); // ok
```

typename

```
struct String
{
    using Char = wchar_t;
};

template <class T>
class Parser
{
    T::Char buffer[]; // Ошибка
};
```

Если компилятор встречая идентификатор в шаблоне, может его трактовать как тип или что-то иное (например, как статическую переменную), то он выбирает иное.

```
struct String
{
    using Char = wchar_t;
};

template <class T>
class Parser
{
    typename T::Char buffer[]; // Ok
};
```

Ошибка инстанцирования шаблона

```
template <class T>
bool lessThen7(T value) { return value < 7; }
```

```
template <class T>
bool lessThen7(T value) { return value < 7; }

struct A {};
A a;
lessThen7<A>(a); //  Инстанцирование
bool lessThen7(A value) { return value < 7; }
// Ошибка инстанцирования, тип a не имеет оператора <
```

SFINAE (Substitution Failure Is Not An Error)

При определении перегрузок функции ошибочные инстанциации шаблонов не вызывают ошибку компиляции, а отбрасываются из списка кандидатов на наиболее подходящую перегрузку.

Неудачная инстанцирование шаблона - это не ошибка.

Например, позволяет на этапе компиляции выбрать нужную функцию:

```
// C++11

template<typename T>
void clear(T& t,
    typename std::enable_if<std::is_pod<T>::value>::type* = nullptr)
{
    std::memset(&t, 0, sizeof(t));
}

// Для не-POD типов
template<typename T>
void clear(T& t,
    typename std::enable_if<!std::is_pod<T>::value>::type* = nullptr)
{
    t = T{};
}
```

enable_if

```
template<bool, typename T = void>
struct enable_if
{
};

// Частичная специализация для true
template<typename T>
struct enable_if<true, T>
{
    using type = T;
};

enable_if<false, int>::type // Ошибка, нет type
enable_if<true, int>::type // Ок, type == int
```

```
template <>
struct is_pod<int>
{
    static constexpr value = true;
};
```

```
// C++14

template<typename T>
void clear(T& t, std::enable_if_t<std::is_pod<T>::value>* = nullptr)
{
    std::memset(&t, 0, sizeof(t));
}

// Для не-POD типов
template<typename T>
void clear(T& t, std::enable_if<!std::is_pod<T>::value>* = nullptr)
{
    t = T{};
}
```

Можно получить на этапе компиляции информацию о типе, например, проверим есть ли у класса некий метод:

```
struct A
{
    void foo() {}
};

struct B
{
};

template<typename T>
struct HasFoo
{
    static constexpr bool value = true;
};

int main()
{
    std::cout << hasFoo<A>::value << '\n';
    std::cout << hasFoo<B>::value << '\n';
    return 0;
}
```

```
template<typename T>
struct HasFoo
{
    static constexpr bool value = ???;
};
```

Нам нужно будет 2 функции: одна принимает класс с нужным нам методом, другая принимает все остальное:

```
template<typename T>
struct HasFoo
{
    // Принимает все
    static int check(...);

    // Принимает нужный нам класс,
    // где есть какая-то foo()
    template <class U>
    static auto check(U* u) -> decltype(u->foo());
};
```

По возвращаемому функцией типу мы поймем, какая из перегрузок была использована, если тип совпадет, то это то, что нам нужно.

Проверка совпадения типов:

```
template <class T1, class T2>
struct IsSame
{
    static constexpr bool value = false;
};

template <class T>
struct IsSame<T, T>
{
    static constexpr bool value = true;
};
```

Финальный вариант:

```
template<typename T>
struct HasFoo
{
private:
    static int check(...);

    template <class U>
    static auto check(U* u) -> decltype(u->foo());

public:
    static constexpr bool value =
        IsSame
        <
            void,
            decltype(HasFoo<T>::check((T*) nullptr))
        >::value;
};
```

```
hasFoo<A>::value == true;
hasFoo<B>::value == false;
```

type_traits

В стандартной библиотеки функции определения свойств типов is_* находятся в заголовочном файле type_traits

Примеры:

```
is_integral  // Является ли тип целочисленным
is_floating_point // Является ли тип типом с плавающей точкой
is_array // Является ли тип типом массива
is_const  // Содержит ли тип в себе квалификатор const
is_pod // Является ли тип POD-типом
has_virtual_destructor // Имеет ли виртуальный деструктор

// И так далее
```

Шаблоны свойств (traits)

```
template <typename T>
struct NumericTraits
{
};

template <> // Специализация
struct NumericTraits<char>
{
    static constexpr int64_t min = -128;
    static constexpr int64_t max = 127;
};
```

```
template <typename T>
class Calculator
{
    T getNumber(const std::string& text)
    {
        const int64_t value = toNumber(text);
        if (value < NumericTraits<T>::min
            || value > NumericTraits<T>::max)
        {
            // range error
        }
    }
};
```

> Смотрите заголовочный файл numeric_limits

Не только значения, но и типы:

```
template <class T>
class BasicStream
{
public:
    using Char = T;
};

using Utf8Stream = BasicStream<char>;

Utf8Stream::Char c;
```

Классы стратегий

Класс стратегий - интерфейс для применения стратегий в алгоритме.

```
class Json
{
public:
    void encode(const char* data, size_t size) {}
};

class Xml
{
public:
    void encode(const char* data, size_t size) {}
};

template <class T, class Format>
class Connector
{
    Format format_;
public:
    void connect()
    {
        auto packet = makeConnectPacket();
        auto encodedPacket = format_.encode(
            packet.data, packet.size);
        send(encodedPacket);
    }
};
```

```
using JsonConnector = Connector<Json>;
```

Отличия между свойствами и стратегиями

Свойство - отличительная особенность характеризующая сущность.

Стратегия - образ действия сущности.

Сравнение динамического и статического полиморфизма

```
class Device
{
public:
    virtual ~Device() {}

    virtual void write(const char* data, size_t size) = 0;
};

class File final
    : public Device
{
public:
    void write(const char* data, size_t size) override {}
};

class Stream
{
    Device* device_;
public:
    explicit Stream(Device* device)
        : device_(device)
    {
    }

    void putChar(char c)
    {
        device_->write(&c, 1);
    }
};

auto stream = Stream(new File("file.txt"));
```

```
class File
{
public:
    explicit File(const char* name) {}
    void write(const char* data, size_t size) {}
};

template <class Device>
class Stream
{
    Device device_;
public:
    explicit Stream(Device&& device)
        : device_(std::move(device))
    {
    }

    void putChar(char c)
    {
        device_.write(&c, 1);
    }
};

using FileStream = BasicStream<File>;

FileStream stream(File("data"));
```

1. Динамический полиморфизм более гибок и позволяет настраивать поведение во время выполнения, но имеет накладные расходы на вызов виртуальных методов
2. Статический полиморфизм не имеет накладных расходов, но менее гибок

Шаблоны с произвольным количеством аргументов (variadic templates)

```
print(1, "abc", 2.5);
```

```
template <class T>
void print(T&& val)
{
    std::cout << val << '\n';
}

template <class T, class... Args>
void print(T&& val, Args&&... args)
{
    std::cout << val << '\n';
    print(std::forward<Args>(args)...);
}
```

Практическая часть

Простой сериализатор поддерживающий два типа: uint64_t и bool.

```
struct Data
{
    uint64_t a;
    bool b;
    uint64_t c;
};

Data x { 1, true, 2 };

std::stringstream stream;

Serializer serializer(stream);
serializer.save(x);

Data y { 0, false, 0 };

Deserializer deserializer(stream);
const Error err = deserializer.load(y);

assert(err == Error::NoError);

assert(x.a == y.a);
assert(x.b == y.b);
assert(x.c == y.c);
```

Сериализовать в текстовый вид с разделением пробелом, bool сериализуется как true и false

Подсказки по реализации

```
struct Data
{
    uint64_t a;
    bool b;
    uint64_t c;

    template <class Serializer>
    Error serialize(Serializer& serializer)
    {
        return serializer(a, b, c);
    }
};
```

```
// serializer.h
#pragma once

enum class Error
{
    NoError,
    CorruptedArchive
};

class Serializer
{
    static constexpr char Separator = ' ';
public:
    explicit Serializer(std::ostream& out)
        : out_(out)
    {
    }

    template <class T>
    Error save(T& object)
    {
        return object.serialize(*this);
    }

    template <class... ArgsT>
    Error operator()(ArgsT... args)
    {
        return process(args...);
    }

private:
    // process использует variadic templates
};
```

Deserializer реализуется аналогично Serializer, только принимает std::istream, а не std::ostream

Пример десериализации bool:

```
Error load(bool& value)
{
    std::string text;
    in_ >> text;

    if (text == "true")
        value = true;
    else if (text == "false")
        value = false;
    else
        return Error::CorruptedArchive;

    return Error::NoError;
}
```

EOF