SFINAE

Пространства типов и значений. Программирование на типах.

К. Владимиров, Synatacore, 2024
mail-to: konstantin.vladimirov@gmail.com

Сложные множества перегрузки

- Шаблоны немного слишком общая штука.

```
// please do not call it for non-trivial T
template <typename T>
T *copy_construct_array(const T* src, unsigned n);
```

- Важной задачей является уточнение этого множества.
- Комментарий помогает плохо. Один из вариантов это выдать явную ошибку…

Сложные множества перегрузки

- Шаблоны немного слишком общая штука.

```
template <typename T>
T *copy_construct_array(const T* src, unsigned n) {
  static_assert(std::is_trivially_copyable_v<T>);
```

- Важной задачей является уточнение этого множества.
- Комментарий помогает плохо. Один из вариантов это выдать явную ошибку…
- Но что такое этот определитель и как сделать свой?

Пространства типов и значений

- Определитель типа отображает тип на значение.

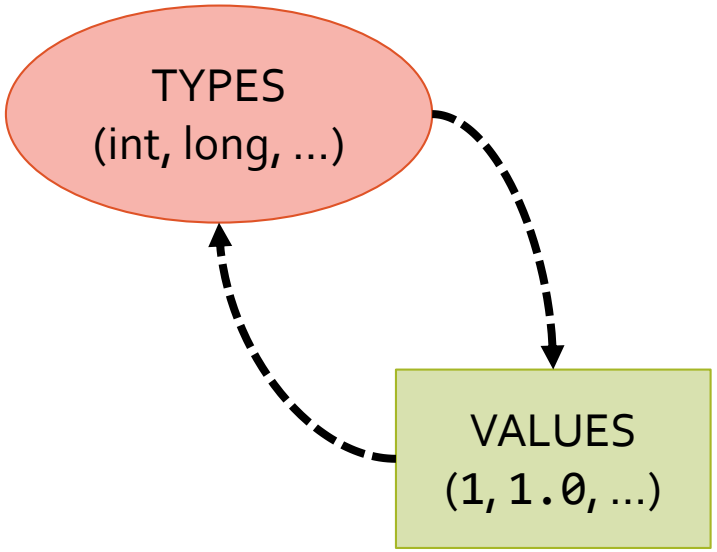
```
std::is_trivially_copyable_v<T>
```

```
T -> (true, false)
```

- Можем ли мы сделать обратное отображение?

```
integral -> T
```

```
(true, false) -> T
```



Отображение значений на типы

- Интегральные константы отображают значения на типы.

```
template <typename T, T v> struct integral_constant {
  static const T value = v;
  typedef T value_type;
  typedef integral_constant type;
  operator value_type() const { return value; }
};
```

- Возможна даже арифметика.

```
using ic6 = integral_constant<int, 6>;
auto n = 7 * ic6{};
```

Отображение типов на true/false

- Каждому из перечисленных ниже типов соответствует одно состояние да или нет.

```
struct True { char c[2]; };
struct False { char c[1]; };
```

- Пример такого рода отображения:

```
template <typename T, typename U> struct is_same : False {};
template <typename T> struct is_same<T, T> : True {};

assert(sizeof(is_same<int,int>) == sizeof(True));
```

- Это не слишком удобно. Можем ли мы придумать лучший подход?

Истина и ложь для типов

- Самые полезные из интегральных констант – самые простые.

```
using true_type = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;
```

- И они же позволяют отображение на истину и ложь.

```
template<typename T, typename U> struct is_same : false_type {};
template<typename T> struct is_same<T, T> : true_type {};
template<typename T, typename U>
using is_same_t = typename is_same<T, U>::type;
```

- Теперь благодаря свойствам частичной специализации, будет работать.

```
assert(is_same<int, int>::value && !is_same<char, int>::value);
```

Польза от шаблонов переменных

- Прошлый слайд может быть доработан

```
template<typename T, typename U> struct is_same : false_type {};
template<typename T> struct is_same<T, T> : true_type {};
template<typename T, typename U>
using is_same_t = typename is_same<T, U>::type;

template<typename T, typename U>
bool is_same_v = is_same<T, U>::value; // не is_same_t<....>::value!
```

- Теперь будет работать как полная, так и сокращённая версия

```
assert(is_same<int, int>::value && !is_same<char, int>::value);
assert(is_same_v<int, int> && !is_same_v<char, int>);
```

Упражнение: логика типов

- Довольно простое упражнение

```
template <typename T, typename U> struct and_ : false_type {};
template <typename T> struct not_ : false_type {};
```

- Необходимо чтобы шаблон `and_` был `true_type` если `T` и `U` вместе `true_type`
- Также необходимо, чтобы шаблон `not_` был `true_type` если `T` это `false_type`

Первая попытка терпит крах

- Мы наивно определяем триады для and_ и not_

```
template <typename T, typename U> struct and_ : false_type {};
template <> struct and_<true_type, true_type> : true_type {};
template <typename T, typename U>
using and_v = and_<T, U>::value;

template <typename T> struct not_ : false_type {};
template <> struct not_<false_type> : true_type {};

assert(and_v<is_same<int, int>, not_<is_same<char, int>>>);
```

- Увы, эта строчка не работает.

https://godbolt.org/z/gnThqvYGc

Типы и значения: unwrap

- Критичный момент это unwrap.

```
using is_same_t = typename is_same<T, U>::type;
```

- По аналогии.

```
template <typename T> struct not_ : false_type {};
template <> struct not_<false_type> : true_type {};
template <typename T> using not_t = typename not_<T>::type;
```

- Теперь всё работает, так как на специализацию попадают верные типы.

```
assert(and_v<is_same<int, int>, not_<is_same<char, int>>>);
assert(and_v<is_same_t<int, int>, not_t<is_same_t<char, int>>>);
```


https://godbolt.org/z/vfPTzTYcG

Определители и модификаторы

Определитель: является ли тип ссылкой.

```
template <typename T> struct is_reference : false_type {};
template <typename T> struct is_reference<T&> : true_type {};
template <typename T> struct is_reference<T&&> : true_type {};
```

Определители и модификаторы

Модификатор: убираем ссылку с типа, если ссылки не было, то оставляем тип.

```
template <typename T>
struct remove_reference { using type = T; };

template <typename T>
struct remove_reference<T&> { using type = T; };

template <typename T>
struct remove_reference<T&&> { using type = T; };
```

- Для модификатора также полезен unwrap алиас.

```
template <typename T>
using remove_reference_t = typename remove_reference<T>::type;
```

Четырнадцать категорий

- Любой тип в языке C++ попадает хотя бы под одну из перечисленных ниже категорий.

```
is_void
is_null_pointer
is_integral, is_floating_point // для T и для cv T& транзитивно
is_array; // только встроенные, не std::array
is_pointer; // включая указатели на обычные функции
is_lvalue_reference, is_rvalue_reference
is_member_object_pointer, is_member_function_pointer
is_enum, is_union, is_class
is_function // обычные функции
```

- Использование довольно тривиально.

```
std::cout << std::boolalpha << std::is_void<T>::value << '\n';
```

Свойства типов

- Также очень полезны определители свойств типов.

```
is_trivially_copyable // побайтово копируемый, memcpy
is_standard_layout // можно адресовать поля указателем
is_aggregate // доступна агрегатная инициализация как в C
is_default_constructible // есть default ctor
is_copy_constructible, is_copy_assignable
is_move_constructible, is_nothrow_move_constructible
is_move_assignable
is_base_of // B является базой (транзитивно, включая сам тип)
is_convertible // есть преобразование из A к B
```

- И многие другие (их реально десятки).

Обсуждение

- Кажется, что добавить lvalue reference совсем просто.

```
template <typename T> struct add_lref { using type = T&; };
```

- Кто видит тут проблемы?

Первая попытка: исключим void

- Увы, это не будет работать для void. Хорошо, добавим void….

```
template <typename T> struct add_lref { using type = T&; };

template <> struct add_lref<void> { using type = void; };

template <> struct add_lref<const void> {
  using type = const void;
};

// то же самое для volatile и const volatile
```

- Но уверены ли мы, что исключать нужно только cv-void?

Что мы на самом деле хотим?

- Увы, это не будет работать для void. Хорошо, добавим void...

```
template <typename T> struct add_lref { using type = T&; };

template <> struct add_lref<void> { using type = void; };
```

- Мы свернули куда-то не туда
- Мы хотим записать: "если можно, то T& иначе T" и эту идею должно быть можно выразить как-то естественным образом.

Вторая попытка: выразим идею ясно

- Возможное решение (via A. O'Dwyer).

```
template <typename T, typename Enable>
struct ALRImpl { using type = T; };

template <typename T>
struct ALRImpl<T, std::remove_reference_t<T&>> { using type = T&; };

template <typename T>
struct add_lref : ALRImpl <T, std::remove_reference_t<T>> {};
```

- Это решение изящно, но всё же тяжеловесно и не очевидно.

Обсуждение: enabled?

- Выбор функции или типа может быть провален при подстановке шаблонного параметра.

```
template <typename T, typename Enable>
struct ALRImpl { .... };

template <typename T>
struct ALRImpl<T, std::remove_reference_t<T&>> { .... };

template <typename T>
struct add_lref : ALRImpl <T, std::remove_reference_t<T>> {};
```

Продолжаем писать copy construct

- Мы остановились на следующей идее.

```
template <typename T>
T *copy_construct_array(const T* src, unsigned n) {
  static_assert(std::is_trivially_copyable_v<T>);
```

- Но что если мы можем это, только иначе?

Что мы думаем о таком варианте?

- В принципе можно добавлять варианты…

```
template <typename T>
T *copy_construct_array(const T* src, unsigned n) {
  auto N = sizeof(T) * n;
  auto *D = static_cast<T*>(::operator new(N));
  if (std::is_trivially_copyable_v<T>)
    D = static_cast<T*>(std::memcpy(D, src, N));
  else if (std::is_nothrow_copy_constructible_v<T>)
    for (int i = 0; i < n; ++i)
      new(&D[i]) T(src[i]);
  // ....
```

- И так далее. Что в этом хуже всего?

Интрузивность

- Мы бы хотели разные функции внутри множества перегрузки.

```
// первый вариант
if (std::is_trivially_copyable_v<T>)
  D = static_cast<T*>(std::memcpy(D, src, N));

// второй вариант
if (std::is_nothrow_copy_constructible_v<T>)
  for (int i = 0; i < n; ++i)
    new(&D[i]) T(src[i]);
```

- Тогда мы бы могли добавлять новые варианты не изменяя существующего кода.

SFINAE

- Substitution Failure Is Not An Error (провал подстановки не является ошибкой).

```
template <typename T> T max(T a, T b); // 1
template <typename T, typename U> auto max(T a, U b); // 2

int g = max(1, 1.0); // подстановка в 1 провалена
                     // подстановка в 2 успешна
```

- Если в результате подстановки в непосредственном контексте класса (функции, алиаса, переменной) возникает невалидная конструкция,
- То эта подстановка неуспешна, но не ошибочна.

SFINAE и ошибки

- Не любая ошибочная конструкция это SFINAE. Важен контекст подстановки.

```
int negate (int i) { return -i; }

template <typename T> T negate(const T& t) {
  typename T::value_type n = -t();
  // тут используем n
}

negate(2.0); // ошибка
```

- В контексте сигнатуры и шаблонных параметров нет никакой невалидности.
- Невалидность в теле не является SFINAE, это ошибка второй фазы трансляции.

SFINAE и ошибки

- Не любая ошибочная конструкция это SFINAE. Важен контекст подстановки.

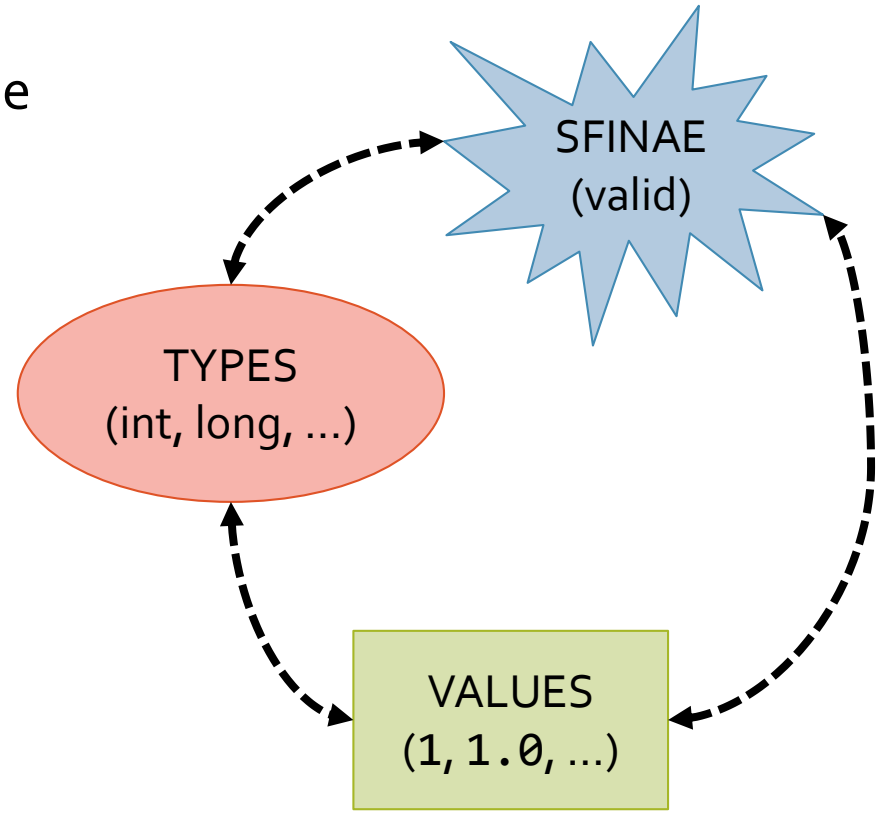
```
int negate (int i) { return -i; }

template <typename T> typename T::value_type negate(const T& t) {
  typename T::value_type n = -t();
  // тут используем n
}

negate(2.0); // substitution failure
```

- Выводится `T` → `double` и, разумеется, `T::value_type` невалидно.
- Здесь нет ошибки, это провал подстановки и будет вызвана менее подходящая нешаблонная функция.

https://godbolt.org/z/rzzo9boWx

Три базовых пространства

- Можно сказать, что в языке существуют отдельные пространства.
  - типов (type-space)
  - значений (value-space)
  - характеристик валидности (sfinae-space)
- Ключевой шаг к тонкому использованию SFINAE это установление их связи.



void_t :: [enabled] -> enabled

- Появился в C++17 как `std::void_t` но довольно прост в реализации.

```
template <typename ...> using void_t = void;
```

- Представляет собой отображение произвольной пачки типов на enabled если каждый из них enabled.

```
template <typename T, typename Enable>
struct ALRImpl { using type = T; };

template <typename T>
struct ALRImpl <T, std::void_t<T&>> { using type = T&; };

template <typename T> struct add_lref : ALRImpl <T, void> {};
```

Условный тип

- Рассмотрим следующее отображение bool -> {T, F}.

```
template <bool B, typename T, typename F>
struct conditional { using type = T; };

template <typename T, typename F>
struct conditional<false, T, F> { using type = F; };

template <bool B, typename T, typename F>
using conditional_t = typename conditional<B, T, F>::type;
```

- Она представляет собой условный тип.

Условный тип без второго условия

- Рассмотрим следующее отображение bool -> {T, F}.

```
template <bool B, typename T, typename F>
struct conditional { using type = T; };

template <typename T, typename F>
struct conditional<false, T, F> { using type = F; };

template <bool B, typename T, typename F>
using conditional_t = typename conditional<B, T, F>::type;
```

- Если сделать его невалидным для F?
- Это станет отображением bool -> {T, invalid}.

enable_if_t :: bool -> enabled

- Получившаяся триада `enable_if` является одной из самых полезных идиом в практическом SFINAE.

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

template <typename T>
struct enable_if<false, T> { };

template <bool B, typename T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

- Она используется, чтобы выкидывать (sfinae-out) инстанциации шаблонов.

Пример SFINAE-OUT

- Например следующая функция инстанцируется только для типов.

```
template <typename T, typename = enable_if_t<(sizeof(T) > 4)>>
void foo(T x) { сделать что-то с x }

foo('c'); // ошибка подстановки
```

- Очевидная проблема: можно ли в пару ей написать функцию для (`sz <= 4`)?

Пример SFINAE-OUT

- Простая идея: написать такую же перегрузку.

```
template <typename T, typename = enable_if_t<(sizeof(T) > 4)>>
void foo(T x) { /* сделать что-то с x */ }

template <typename T, typename = enable_if_t<(sizeof(T) <= 4)>>
void foo(T x) { /* сделать что-то ещё с x */ }

foo('c'); // ошибка разрешения перегрузки
```

- Простая идея не работает. Но почему?

https://godbolt.org/z/sEP79Pz5G

ODR важнее всего остального

- Компилятор не знает, что один из этих шаблонов даст ошибку подстановки.
- С точки зрения генерации ассемблерных меток, важные части сигнатуры выглядят одинаково.
- Параметры по умолчанию в сигнатуре не участвуют

```
template <typename T, typename U = нечто>
void foo(T x) { нечто }

template <typename T, typename U = нечто>
void foo(T x) { нечто }
```

- Понятно, что это техническое нарушение ODR.

Dummy параметр

- Можно выкрутиться с dummy-аргументом.

```
template <typename T, typename = enable_if_t<(sizeof(T) > 4)>>
void foo(T x) { /* сделать что-то с x */ }

template <typename T, typename = enable_if_t<(sizeof(T) <= 4)>>
void foo(T x, int dummy = 0) { /* сделать что-то ещё с x */ }

foo('c'); // ok, but pain
```

- Это работает, но это странная ассиметрия на ровном месте.

Указатель в шаблоне

- Разумное решение это изменить сигнатуру.

```
template <typename T, enable_if_t<(sizeof(T) > 4)>* = nullptr>
void foo(T x) { /* сделать что-то с x */ }

template <typename T, enable_if_t<(sizeof(T) <= 4)>* = nullptr>
void foo(T x) { /* сделать что-то ещё с x */ }

foo('c'); // OK
```

- Теперь компилятор должен ждать подстановки чтобы оценить сигнатуру.
- Заметьте в отличие от void_t тут всё срастается идеально.

https://godbolt.org/z/rjKq5xodb

Целочисленный аргумент там же

- Упростим немного.

```
template <typename T, enable_if_t<(sizeof(T) > 4), int> = 0>
void foo(T x) { /* сделать что-то с x */ }

template <typename T, enable_if_t<(sizeof(T) <= 4), int> = 0>
void foo(T x) {/* сделать что-то ещё с x */ }

foo('c'); // OK?
```

- Будет ли это работать?

https://godbolt.org/z/dvTrxc719

Некоторые проблемы void_t

- Пусть нам нужно сделать перегрузку по SFINAE-cases

```
template <typename T, std::void_t<typename T::x>* = nullptr>
void foo(T x) { /* сделать что-то с x */ }

template <typename T, std::void_t<typename T::y>* = nullptr>
void foo(T x) {/* сделать что-то ещё с x */ }
```

- В данном случае это не сработает потому что void_t это алиас.

https://godbolt.org/z/c79EWe7aK

Правила для алиасов

- Полная и частичная специализация шаблонов алиасов невозможна.

```
template <typename T> using MyType = std::vector<T>;
template <> using MyType<int> = int; // ошибка
```

- Алиасы шаблонов не выводятся выводом типов:

```
template<class T> using Vec = std::vector<T>;

Vec<int> v; // std::vector<T, std::allocator<T>> v;

template<template<typename> typename TT> void f(TT<int>);

f(v); // ошибка вывода
```

- Алиасы шаблонов исчезают почти сразу.

Трюк Хаггинса

- Небольшая материализация void_t позволяет решить проблему.

```
template <typename T> struct TypeIdentity {
  using type = T;
};

template <typename ...> struct Void : TypeIdentity<void> {};

template <typename ... Args> using Void_t =
  typename Void<Args...>::type;
```

- Теперь алиас это прямой не алиас на void, это алиас на зависимый тип.
- Несмотря на подстановку алиаса, мы будем ждать инстанцирования для подстановки типа.

https://godbolt.org/z/odr3d9aTr

Обсуждение

- Настоящее место SFINAE: выкрутиться там, где не хватает средств языка.

Case study: one-off function

- К сожалению, в стандартной библиотеке `std::function` требует копируемости замыкания.

```
auto n = std::make_unique<int>(42);
std::function<ftype> f = [nrr = std::move(n)] { /* .... */ };
```

```
error: use of deleted function
'main()::<lambda()>::<lambda>(const main()::<lambda()>&)'
```

- Предположим, что вы хотите написать свой класс функции, который позволяет такие вещи.
- Разумеется он не будет копируемым, но в целом это может быть и не нужно.

Case study: one-off function

- Начать просто.

```
template <typename T> class fire_once;

template <typename R, typename... Args> class
fire_once<R(Args...)> {
  using anydeleter = void (*)(void *);
  unique_ptr<void, anydeleter> ptr {nullptr, +[](void *) {}};
  R (*invoke)(void *, Args...) {nullptr};

public:
  // TODO: operator()
```

- Опустим не слишком элегантный конструктор. Как написать оператор вызова?

Case study: one-off function

- Первая попытка: наивный оператор вызова.

```
template <typename R, typename... Args> class
fire_once<R(Args...)> {
  // unique_ptr<void, anydeleter> ptr;
  // R (*invoke)(void *, Args...);

public:
  R operator()(Args... args) {
    R ret = invoke(ptr.get(), args...);
    clear(); // очистка: invoke = nullptr, ptr.reset()
    return ret;
  }
```

- Тут есть некоторые проблемы.

Обсуждение

- В языке есть два вида функций: те которые возвращают void и все остальные.

```
template <typename T> void foo(T t) {}

fire_once f = [nrr = std::move(n)] { foo(std::move(nrr); }
```

- Тут очевидная ошибка компиляции.

```
R operator()(что-то) { // теперь R == void
  R ret = чему-то;
```

- Но кажется мы знаем что делать: всего-то разграничить по возвращаемому типу с помощью SFINAE.

Попытка решения

- Попробуем вычеркнуть лишнее через SFINAE.

```
template <enable_if_t<!is_same<R, void>{}, int> = 0>
R operator()(Args... args) && {
  R ret = invoke(ptr.get(), args...); clear(); return ret;
}

template <enable_if_t<is_same<R, void>{}, int> = 0>
void operator()(Args... args) && {
  invoke(ptr.get(), args...); clear();
}
```

- Будет ли это работать?

https://godbolt.org/z/cs4q0E3Ye

Обсуждение

- Мы добрались до действительно странных мест

```
error: no type named 'type' in 'std::enable_if<false, int>';
'enable_if' cannot be used to disable this declaration
```

- Что нам здесь говорят?
- Такое чувство, что компилятор даже понимает что мы хотим сделать. Просто не может этого сделать.
- Ключ к ответу: ленивость и энергичность. Будет ли в данном случае проверка отложена до подстановки шаблонного параметра?

Настоящее решение

- Это называется "трамплинным SFINAE". Мы заводим специальный тип для разрешения которого нужно ждать точку инстанцирования

```
template <typename R2 = R,
          enable_if_t<!is_same<R2, void>{}>* = nullptr>
R2 operator()(Args... args) && {
  R2 ret = invoke(ptr.get(), args...); clear(); return ret;
}

template <typename R2 = R,
          enable_if_t<is_same<R2, void>{}>* = nullptr>
R2 operator()(Args... args) && {
  invoke(ptr.get(), args...); clear();
}
```


https://godbolt.org/z/5fr4zYPcq

Обсуждение

- Конструкция conditional_t (и её частный случай enable_if) напоминают условный оператор в неизвестном языке программирования

Открытие метапрограммирования

```
// Prime number computation by Erwin Unruh
template <int i> struct D { D(void*); operator int(); };

template <int p, int i> struct is_prime {
  enum { prim = (p % i) && is_prime<(i > 2 ? p : 0), i -1> :: prim };
};

template <int i> struct Prime_print {
  Prime_print<i-1> a;
  enum { prim = is_prime<i, i-1>::prim };
  void f() { D<i> d = prim; }
};

struct is_prime<0, 0> { enum { prim = 1 }; };
struct is_prime<0, 1> { enum { prim = 1 }; };
struct Prime_print<2> { enum { prim = 1 }; void f() { D<2> d = prim; } };

main () { Prime_print<10> a; }
```

Обсуждение

- По аналогии с партизанским SFINAE, это выглядит партизанским метапрограммированием. Тут с путей сходят целые составы.
- Сама идея метапрограммирования увлекательна: мы делаем всякие вещи на этапе компиляции.
- Нельзя ли придумать более систематичный подход к такой технике?

Факториал

- Идея лежит на поверхности: что если развернуть систематическое sfinae от типов на целые числа?

```
template<size_t N>
struct fact : integral_constant<size_t, N * fact<N - 1>{}> {};

template<> struct fact<0> : integral_constant<size_t, 1> {};
```

```
cout << fact<5>::value << endl;
```

- Например инстанцирования в этом примере легко проследить

Факториал

- Идея лежит на поверхности: что если развернуть систематическое sfinae от типов на целые числа?

```
template<size_t N>
struct fact : integral_constant<size_t, N * fact<N - 1>{}> {};

template<> struct fact<0> : integral_constant<size_t, 1> {};

fact<1> : integral_constant<size_t, 1 * fact<0>{}> // → 1
fact<2> : integral_constant<size_t, 2 * fact<1>{}> // → 2
fact<3> : integral_constant<size_t, 3 * fact<2>{}> // → 6
fact<4> : integral_constant<size_t, 4 * fact<3>{}> // → 24
fact<5> : integral_constant<size_t, 5 * fact<4>{}> // → 120
cout << fact<5>::value << endl; // → 120
```

- Умножение работает за счёт наличия operator size_t().

Числа Фибоначчи

- С той же лёгкостью можно вычислять на этапе компиляции числа Фибоначчи.

```
template<size_t N>
struct fibonacci :
     integral_constant< size_t,
                        fibonacci<N-1>{} +
                        fibonacci<N-2>{}> {};
template<> struct fibonacci<1> : integral_constant<size_t,1> {};
template<> struct fibonacci<0> : integral_constant<size_t,0> {};
```

- Не смущает ли нас здесь двойная рекурсия?

Две модели вычислений

- "Императивная"

```
int fact_0 (int x) {
  int i = 2, res = 1;
  for (; i <= x; ++i)
    res *= i;
  return res;
}
```

- Временные переменные.
- Циклы.
- Изменяемая память.

- "Функциональная"

```
const int fact_1 (const int x) {
  if (x < 2)
    return x;
  else
    return x * fact_1 (x - 1);
}
```

- Вызовы функций.
- Рекурсия.
- "Чистые" вычисления.

➢ Как вы предпочтёте написать функцию, вычисляющую факториал и почему?

Целочисленный квадратный корень

- Чтобы делать такие сложные вещи на шаблонах, полезно сначала просто написать программу в функциональном стиле.

```
int isqrt (int N, int lo = 1, int hi = N) {
  int mid = (lo + hi + 1) / 2;
  if (lo == hi)
    return lo;
  else {
    if (N < mid * mid)
      return isqrt (N, lo, mid - 1);
    else
      return isqrt (N, mid, hi);
  }
}
```

Целочисленный квадратный корень

- Рассмотренный ранее conditional_t вполне сработает в качестве meta-if.

```
template <int N, int L = 1, int H = N, int mid = (L + H + 1) / 2>
struct Sqrt : integral_constant<int,
                conditional_t<(N < mid * mid),
                              Sqrt<N, L, mid - 1>,
                              Sqrt<N, mid, H> >{}> {};

template <int N, int S> struct Sqrt <N, S, S, S> :
  integral_constant<int, S> {};
```

- Домашняя наработка: попробуйте найти N-е простое число на этапе компиляции в таком стиле.

https://godbolt.org/z/fExhs7rjd

Обсуждение

- Разумеется, арифметические метапрограммы это забивание гвоздей микроскопом. Уже скоро constexpr функции будут делать это гораздо лучше.
- Но изучение арифметических метапрограмм – прекрасная тренировка ума для осознания настоящего метапрограммирования на типах.

Loki

- В 2001 году Андрей Александреску опубликовал [$MCPP$].
- Ключевая идея в книге это идея **списка типов**.

```
template <class T, class U>
struct TypeList {
  using Head = T;
  using Tail = U;
};

using Types = LOKI_TYPELIST_4(int, long, float, double);

using Reversed = Loki::TL::Reverse<Types>::Result;

using Third = Loki::TL::TypeAt<Reversed, 3>::Result; // long
```

Boost::MPL

- Дальнейшее развитие метапрограммирования это MPL (см. также [$MPLBook$]).
- Эта библиотека содержала контейнеры типов и алгоритмы для работы с ними.

```
using Types = mpl::vector<int, long, float, double>;

using Ptrs = mpl::transform<Types,
                            std::add_pointer<mpl::_1>>::type;
```

- По сути здесь систематическое SFINAE выраженное через метапрограммирование.
- Это делало работу со списками типов ещё проще.

https://godbolt.org/z/nsqsefYzq

Boost::Fusion

- Около 2008 года, сильно обогнав своё время появилась библиотека, которая позволяла работать гетерогенным образом и с типами и со значениями.

```
auto to_string =
  [](auto t) { stringstream ss; ss << t; return ss.str(); };

fusion::vector<int, std::string, float> seq {1, "abc", 3.4f};
auto noflts =
        fusion::remove_if<std::is_floating_point<mpl::_>>(seq);
auto strs = fusion::transform(noflts, to_string);
           // → fusion::vector<string, string>
```

- Обратите внимание на то, что по сути здесь речь о `std::tuple` за три года до его стандартизации.

Boost::Fusion

- Я не удержусь от ещё одного примера.

```
using fv = fusion::vector<int, std::string, bool, double>;

fv seq {1, "abc", 3.4f};

auto seq2 = fusion::push_back(seq, 'X');

cout << size(seq) << " " << back(seq) << " "
     << size(seq2) << " " << back(seq2) << endl;
```

- Может показаться, что это уже не совсем метапрограммирование.
- Но это как раз именно оно.

Абстракция метрики

- В своём $[MCF]$, Майкл Кейси приводит следующий пример вычисления Евклидова расстояния.

```
template <typename P1, typename P2>
double pydist(P1 p1, P2 p2) {
  typedef fusion::vector<P1&, P2&> zip_t;

  // вычисляется сумма квадратов разностей
  double accumulated = fusion::fold(
      fusion::zip_view<zip_t>(zip_t(p1, p2)), 0, pythagoras()
  );

  return sqrt(accumulated);
}
```

Функтор для метрики

- Использованный функтор разумеется также сделан через fusion.

```
struct pythagoras {
  typedef double result_type;

  template<typename T>
  double operator()(double acc, T const & axis) const {
    double d = fusion::at_c<0>(axis) - fusion::at_c<1>(axis);
    return acc + d * d;
  }
};
```

- Но что удивительно, эти вычисления можно проводить не только для специальных, а по сути для любых структур.

Адаптация структур

- В силу своего магического шаблоноподобного действия, fusion требует структуры как списка типов.
- Но обычную структуру можно адаптировать для его алгоритмов.

```
struct mypoint {
  double x = 0.0, y = 0.0;
};

BOOST_FUSION_ADAPT_STRUCT(
  mypoint,
  (double, x)
  (double, y)
);
```


https://godbolt.org/z/88afarc09

Обсуждение

- Что дальше?
- Дальше наступает наше время и принимают стандарт C++11
- И всё становится совсем окончательно интересно

Литература

- ISO/IEC, "Information technology – Programming languages – C++", ISO/IEC $14882{:}2017$
- Bjarne Stroustrup – The C++ Programming Language (4th Edition)
- $[MCPP]$ Andrei Alexandrescu, Modern C++ Design. Generic programming and design patterns applied, 2001
- $[MPLBook]$ Abrahams D., Gurtovoy A. – C++ Template Metaprogramming Concepts, Tools, and Techniques from Boost, 2004
- $[MCF]$ Michael Caisse: Solving World Problems with Fusion, CppNow'2013
- Davide Vandevoorde, Nicolai M. Josuttis – C++ Templates. The Complete Guide, 2017
- Louis Dionne, "Metaprogramming in C++14", ACCU'2017
- Arthur O'Dwyer "A Soupçon of SFINAE", CppCon'2017