Чепурні Мультиметоди для Сучасного C++

Анотація

Мультиметоди, або ж множинна диспетчеризація, це механізм вибору однієї з декількох функцій в залежності від динамічних типів або значень аргументів[1]. Потреба в такому механізмі виникає, наприклад, в архітектурних рішеннях, де численні класи взаємодіють між собою у специфічний для кожної пари спосіб. C++ на рівні мови не підтримує такий механізм а пропозиції що до розширення C++ такими інструментами[2] (поки що?) не включені до попереднього плану C++23. Диспетчеризацію можна реалізувати за допомогою комбінації `std::visit` та `std::variant` (див [11], [15]). Ця стаття пропонує альтернативний підхід, заснований на користувацькій ідентифікації і інтроспекції типів, та досліджує різноманітні техніки сучасного C++.

Метою цієї статті не є просування готового рішення, що підійшло б усім і на усі випадки. Натомість у цій статті ми з вами розглянемо які підходи можна використати для різних сценаріїв використання.

1 Вступ

1.1 Існуючі Рішення

Оскільки потреба в множинній диспетчеризації стара як світ програмування, створено чимало рішень з використанням існуючих інструментів C++ (див. [3], [4], [5], [6], [7], [8], [9], [10]). Проте жодне з них не має чепурного вигляду. Безперечно, чепурність – категорія суб'єктивна і, на думку автора, чепурне рішення це щось просте, сучасне, необтяжливе, неінтрузивне та структуроване. Тобто складність такого рішення відповідає складності задачі, його використання не привносить додаткових залежностей та не ускладнює підтримку коду, не вимагає істотних змін у дизайні проєкту, і кожен елемент рішення вирішує одну потребу[12] чи відповідає за один обов'язок[13].

1.2 Огляд Проблем

Для динамічної, часу виконання, диспетчеризації потрібно знати тип об'єкту. C++ має механізми RTTI які можна використати для тих проєктів, що вже використовують RTTI чи можуть собі дозволити таку залежність. Проте рішення не буде чепурним, якщо не надасть альтернативу для проектів, де використання RTTI було б занадто обтяжливим.

Традиційно, подвійна диспетчеризація виконується через віртуальний метод, у якому реалізована ручна диспетчеризація наступного рівня через оператори `if` чи `switch`. Хоча, з точки зору моделювання даних, ці методи переважно не є природною частиною класів, де вони реалізовані, а радше визначають зовнішні операції над класами об'єктів. То ж вони були б більш органічно представлені як функції[2]. Проте ми не будемо відкидати можливість використання методів.

1.3 Моделі Даних

У цій статті ми будемо вправлятись на ієрархії фігур та ієрархії виразів калькулятора із простим успадкуванням:

```
namespace shapes {
struct Shape {
	virtual ~Shape() {}
};
struct Rect : Shape { };
struct Circle : Shape {};
struct Square : Rect { };
}
namespace calculus {
struct Expression {
	virtual ~Expression() {}
};
struct Constant : Expression {};
struct Integer : Constant {};
struct Float : Constant {};
}
```

2 Мультидиспетчер Функцій

Почнемо із моделі `calculus`. Припустимо, що визначені операції `add`, `sub`, і ці операції реалізовані як шаблонні функції:

```
template<class A, class B>
Expression* add(const A&, const B&);
template<class A, class B>
Expression* sub(const A&, const B&);
```

Це дещо спростить перший крок, а ускладнювати будемо поступово.

Отже, наш мультиметод має з'ясувати фактичні типи аргументів, базуючись на цих типах вибрати потрібну функцію із наявних, та викликати обрану функцію. Тут ми окреслили три потреби. Розглянемо їх по черзі.

2.1 Список Функцій

Вибір функції із наявних. Для створення списку цих функцій використаємо варіадичний шаблон із auto параметрами, щоб використати можна було таким чином:

```
multidispatcher<
  add<Integer, Integer>,
  add<Float, Integer>,
  add<Integer, Float>,
  add<Float, Float>>::dispatch(a,b);
```

Для такого використання наш шаблон має виглядати десь так:

```
template<auto Entry, auto ... Entries>
struct multidispatcher {
  template<class ... Arguments>
  static auto dispatch(Arguments& ... arguments) {
    if (entry<Entry>::matches(arguments...)) {
      return entry<Entry>::call(arguments...);
    }
    if constexpr (sizeof...(Entries)>0) {
      return multidispatcher<Entries...>::dispatch(arguments...);
    }
  }
};
```

У методі **dispatch** ми делегували визначення фактичного типу та виклик функції іншому шаблону `entry`, давайте його напишемо:

```
template<auto Method>
struct entry;

template<class Return, class ... Parameter, Return (*Function)(Parameter...)>
struct entry<Function> {
  template<class ... Argument>
  static constexpr bool matches(const Argument& ... argument) noexcept {
    return (expected<Parameter>::matches(argument) and ...);
  }
  template<class ... Argument>
  static Return call(Argument& ... argument) noexcept(noexcept(Function)) {
    return (*Function)((Parameter)(argument)...);
  }
};
```

Шаблон `expected`, який ми тут використали, буде визначати тип фактичного аргументу та чи відповідає цей тип очікуваному. Таким чином повна реалізація набуде такого вигляду

```
template<class Parameter>
struct expected {
  template<class Argument>
  static constexpr bool matches(const Argument& argument) noexcept {
    return typeid(Parameter) == typeid(argument);
  }
};
template<auto Method>
struct entry;

template<class Return, class ... Parameter, Return (*Function)(Parameter...)>
struct entry<Function> {
  template<class ... Argument>
  static constexpr bool matches(const Argument& ... argument) noexcept {
    return (expected<Parameter>::matches(argument) and ...);
  }
  template<class ... Argument>
  static Return call(Argument& ... argument) noexcept(noexcept(Function)) {
    return (*Function)((Parameter)(argument)...);
  }
};

template<auto Entry, auto ... Entries>
struct multidispatcher {
  template<class ... Arguments>
  static auto dispatch(Arguments& ... arguments) {
    if (entry<Entry>::matches(arguments...)) {
      return entry<Entry>::call(arguments...);
    }
    if constexpr (sizeof...(Entries)>0) {
      return multidispatcher<Entries...>::dispatch(arguments...);
    }
  }
};
```

Поекспериментувати із цією реалізацією можна за посиланням: godbolt.org/z/oz585K

Реалізація вийшла доволі проста, проте, порівнюючи із open methods, вона має певні обмеження:

1) довжина списку функцій обмежена максимальною глибиною рекурсії компілятора
2) не підтримується коваріантність результату
3) не підтримується коваріантність параметрів
4) вибирається не найкраща як в open method, а перша-ліпша функція
5) немає поділу на virtual та звичайні параметри – усі очікуються virtual

Через друге обмеження ми не можемо використати функцію що повертає коваріантний результат, наприклад таку: `Float* sub(const Float&, const Float&)`. Третє, четверте та п'яте обмеження дещо звужують сферу сценаріїв використання. У одному із наступних розділів ми розглянемо як їх обійти. Поки що повернемось до більш критичних обмежень 1 та 2.

3 Мультиметоди

3.1 Коваріантність Результату

Щоб підтримати коваріантність результату нам необхідно задати найбільш загальний тип результату із усіх використаних у функціях. Створимо новий шаблон із додатковим параметром – типом результату:

```
template<typename ReturnType, auto ... Entries>
struct multimethod;
```

Раз у нас вже є тип результату, ми можемо використати його також щоб замінити рекурсію на згортку параметрів шаблону:

```
template<typename ReturnType, auto ... Entries>
struct multimethod {
```

```
  template<class ... Arguments>
  static auto dispatch(Arguments& ... arguments) {
    ReturnType value;
    if (((entry<Entries>::matches(arguments...)
      and ((value = entry<Entries>::call(arguments...)),true)) or ...))
      return value;
    else
      throw std::logic_error("Missing dispatch entry");
  }
};
```

Лінк на модифікований приклад: godbolt.org/z/rj5vvY

Зазначимо, що такий підхід потребує default constructible тип результату та підтримки ним move семантики. Для нашого прикладу це не суттєво, проте для інших сценаріїв використання слід враховувати цю особливість.

Допитливий читач можливо звернув увагу що в останньому прикладі не використовується перевантаження функції **sub**:

```
Float* subf(const Float&, const Float&);
Integer* subi(const Integer&, const Integer&);
```

Це через особливості C++ для перевантажених функцій. Щоб отримати адресу такої функції потрібно задати її тип: `(Float* (*)(const Float&, const Float&))&sub`. Цей незручний синтаксис можна спробувати трохи підсолодити шаблоном `resolve`: `resolve<Float*, const Float&, const Float&>{}(&sub)`.

3.2 Заміна RTTI

Розглянемо тепер як можна обійти залежність від RTTI, що виникла через наше використання **typeid**. Для того щоб підтримати сценарії використання без RTTI, прикладна модель має реалізовувати якийсь механізм користувацької ідентифікації та інтроспекції типів (КІІТ). Наприклад, це може бути призначений вручну чи автогенерований числовий ID для кожного класу. Для автогенерації можна використати геш від `__PRETTY_FUNCTION__`:

```
template<class Class>
constexpr auto class_hash() noexcept {
  return hash<>(std::string_view(__PRETTY_FUNCTION__));
}
```

Нажаль, `std::hash` ще не є **constexpr**, тож доведеться написати і свою геш функцію (наприклад, таку як в [14]). Тепер ми можемо призначати ID нашим класам:

```
  static constexpr auto classid = class_hash<Rect>();
```

Для доступу до *classid* напишемо шаблонну функцію **classinfo**:

```
using class_info = size_t;
template<class Class>
class_info classinfo() noexcept {
  return Class::classid;
}
```

А для динамічної інтроспекції типів створимо віртуальний метод :

```
  //Shape
  virtual bool instance_of(const class_info& expected) const noexcept {
    return classinfo<decltype(*this)>() == expected;
  }

  //Rect, Circle
  bool instance_of(const class_info& expected) const noexcept override {
    return classinfo<decltype(*this)>() == expected or Shape::instance_of(expected);
  }
```

А щоб його використання не створювало залежності від користувацького типу `class_info`, заховаємо його за таким фасадом:

```
  //Shape
  template<class Expected>
  bool instanceof() const noexcept {
    return instance_of(classinfo<Expected>());
  }
```

Зверніть увагу, що ці методи не перевантажені. Це спростить допоміжний шаблон **has_instanceof**.

Поки ми використовували RTTI, можна було ігнорувати відмінності між віртуальними і звичайними параметрами - **typeid** працює із усіма типами, а додаткова перевірка на співпадіння статичних типів прибирається із вихідного коду оптимізатором. Проте із КІІТ нам слід вирішити як розпізнати невіртуальні параметри та як порівнювати типи для них. Для розпізнавання можна встановити таке правило: параметри поліморфічних типів, що передаються як lvalue reference є віртуальними, усі інші - не є такими:

```
template<class Class>
struct is_virtual_parameter {
  static constexpr bool value =
       std::is_polymorphic_v<std::remove_reference_t<Class>> and
       std::is_reference_v<Class>;
};
```

А для перевірки невіртуальних параметрів це може бути, наприклад, is_same, чи is_assignable. Виходячи з такого дизайну КІІТ, модифікуємо наш шаблон `expected`:

```
template<class Parameter>
struct expected {
  using parameter_type = std::remove_reference_t<std::remove_cv_t<Parameter>>;
  template<class Argument>
  static constexpr bool matches(Argument&& argument) noexcept {
    if constexpr(has_instanceof<Argument>(0)) {
      return argument.template instanceof<parameter_type>();
    } else {
#if __cpp_rtti >= 199711
      return typeid(Parameter) == typeid(argument);
#else
      static_assert(not is_virtual_parameter_v<Parameter>, "No class info available");
      return is_assignable_parameter_v<Parameter, Argument>;
#endif
    }
  }
};
```

3.3 Підтримка Методів

Поки що наш `multimethod` шаблон підтримує лише функції. Напишемо підтримку для методів. Перше що необхідно - це відділити об'єкт від решти аргументів, наприклад створивши новий метод що б приймав об'єкт першим аргументом:

```
// multimethod
template<class Target, class ... Arguments>
static auto call(Target& target,Arguments& ... arguments) {
  ReturnType value;
  if (((entry<Entries>::matches(target, arguments...)
    and ((value = entry<Entries>::call(target, arguments...)),true)) or ...))
    return value;
  else
    throw std::logic_error("Dispatcher failure");
}
```

Та додати спеціалізацію entry для методів:

```
template<class Target, class Return, class ... Parameter, Return
(Target::*Method)(Parameter...)>
struct entry<Method> {
  template<class Object, class ... Argument>
  static constexpr bool matches(Object& obj, Argument& ... argument) noexcept {
    return expected<Target>::matches(obj)
       and (expected<Parameter>::matches(argument) and ...);
  }
  template<class Object, class ... Argument>
  static Return call(Object& target, Argument& ... argument) {
    return ((Target&)(target).*Method)((Parameter&)(argument)...);
  }
};
```

З методами трохи складніше ніж із функціями, бо їх сигнатура може включати також специфікатор const. Тож знадобиться дві спеціалізації нашого entry, та два методи call. З такою реалізацією наш multimethod дозволяє використовувати методи поряд із функціями. Код для цього прикладу доступний за посиланням: godbolt.org/z/ehEnnc

3.4 Коваріантність Параметрів

Завдяки нашій реалізації КІІТ з викликом instance_of базового класу ми також вирішили проблему коваріантності параметрів. Проте щоб вона працювала як слід, першими у списку мають іти функції/методи із більш специфічними параметрами. Впорядкувати список під час компіляції не видається можливим. Натомість, можна додати перевірку часу компіляції – перевіряти чи далі по списку функцій відсутні такі що приймають породжений від поточного клас. Складність такої перевірки можна оцінити як $O(kn^2)$ де k кількість параметрів а n - кількість функцій. Слід зауважити, що на великих списках така перевірка може уповільнити компіляцію.

3.5 Множинне Успадкування

Також, із цією реалізацією КІІТ можна вирішити проблему із множинним успадкуванням – класу, що успадковує кілька базових класів, достатньо викликати instance_of усіх його базових класів. Однак, можливі випадки, коли простого впорядкування списку виявиться недостатньо для вибору найкращого кандидата для певних комбінацій типів вхідних параметрів.

3.6 Можливі Вдосконалення

Крім того підтримка довгих списків функцій може стати занадто обтяжливою. Щоб спростити роботу із великою кількістю функцій можна розбити список на групи по однотипному першому параметру, наприклад так:

```
groupdispatcher<
  group<Expression*,
    add<Integer, Float>,
    add<Integer, Integer>>,
  group<Expression*,
    add<Float, Integer>,
    add<Float, Float>>>::dispatch(a,b);
```

Можна також повернутись до типового рішення подвійної диспетчеризації через віртуальні методи у якому використати наші чепурні шаблони для наступного рівня диспетчеризації.

3.7 Продуктивність

В такій реалізації multimethod послідовно перебирає кандидати поки не знайде найкращий. В компільованих прикладах на один елемент списку приходиться порядку 10 інструкції. Експерименти показують в середньому 700 інструкцій для таблиці із 40 функції (див приклад perf.cpp або godbolt.org/z/1caGTs). Табличний диспетчинг, очікувано, показав би кращий результат.

4 Табличний Диспетчинг

4.1 Припущення

Для табличного, а для двох параметрів - це матричний, потрібно виконання певних вимог та розв'язання деяких викликів:

1. Для ефективної диспетчеризації ідентифікатори класів мають бути послідовними, та без пропусків. Ручне присвоєння ідентифікаторів, для деяких проєктів, може видатись занадто обтяжливим.
2. Type-safe матриця може містити тільки функції одного типу, тобто із однаковою сигнатурою. Ручне заповнення такої матриці не стійке до помилок, обтяжливе навіть для невеликої кількості класів, а для великої – практично не здійсненне.

4.2 Дизайн Матриці

Проблеми ми окреслили, спробуємо знайти їм рішення. Виходимо, як і в попередніх розділах, із того що для ієрархії з N класів задано K функцій, які ми можемо перелічити у нашому варіадичному шаблоні:

```
matrixdispatcher<
  add<Integer, Integer>,
  add<Float, Integer>,
  add<Integer, Float>,
  add<Float, Float>>::dispatch(a,b);
```

Для початку, припустимо що послідовна ідентифікація класів виконана користувачем:

```
  static constexpr auto classid = 1;
  virtual size_t classID() const { return classid; }
```

Якщо вимагати від функції однакову сигнатуру, наш шаблон, вибираючи кандидата, не зможе відрізнити одну від іншої. Також, така вимога однакової сигнатури перенесла б відповідальність down cast на користувача. Отже нам потрібні проміжні функції що мали б однакову сигнатуру. Завдяки шаблонам, ми можемо доручити компілятору згенерувати потрібну кількість функцій, та заповнити матрицю вказівниками на ці функції. Виведення такої сигнатури із списку отриманих функції не видається можливим. Тож дозволимо користувачу задати цей тип вхідним параметром `FunctionType` шаблона. Також ми не можемо визначити кількість класів – і цю інформацію отримаємо від користувача.

Таким чином, наша матриця могла б мати вигляд як у наступному прикладі:

```
FunctionType matrix[sizeA][sizeB];
```

Проте заповнювати такий масив не дуже зручно. Наприклад його не можна повернути із функції. Тому дещо ускладнимо визначення нашої матриці:

```
std::array<std::array<FunctionType, sizeA>, sizeB> matrix;
```

Тепер пошукаємо рішення для її `constexpr` заповнення вказівниками на згенеровані функції. `constexpr` необхідний щоб гарантувати, що це заповнення буде відбуватись під час компіляції. C++ має засоби для заповнення `constexpr` масивів – це `index_sequence`. Наприклад, заповнення одного рядка нашої таблиці можна досягти таким кодом:

```
std::array<FunctionType,N> { &Template<I> ... } {}
```

Де `Template` – це шаблонна функція, що приймає число як її параметр, а `I` – послідовність індексів. З таким дизайном, шаблон для рядка нашої матриці набуде вигляду:

```
template<template<size_t> class Template, size_t N>
class jumpvector : public std::array<typename Template<0>::value_type,N> {
public:
  using value_type = typename Template<0>::value_type;
  constexpr jumpvector() : jumpvector<Template,N>(std::make_index_sequence<N>()) {}
private:
  template<size_t ... I>
```

```
  constexpr jumpvector(std::index_sequence<I...>)
    : std::array<value_type,N> { &Template<I>::function ... } {}
};
```

Тут довелось перейти від шаблону функції `Template`, сигнатура якої наперед невідома, до шаблону класу `Template` що має статичний метод `function`. Застосувавши цей підхід до рядків, можна заповнити усю матрицю:

```
template<template<size_t, size_t> class Template, size_t N, size_t M>
class jumpmatrix : public std::array<std::array<typename Template<0,0>::value_type, M>,
N> {
public:
  using value_type = std::array<typename Template<0,0>::value_type, M>;
  constexpr jumpmatrix() : jumpmatrix<Template, N, M>(std::make_index_sequence<N>()) {}
private:
  template<size_t ... I>
  constexpr jumpmatrix(std::index_sequence<I...>)
    : std::array<value_type, N> {
        jumpvector<reduce<Template,I>::template type,M>{} ... } {}
};
```

Таким чином визначилися вимоги до шаблону функції для заповнення матриці:

```
  template<size_t A, size_t B>
  struct func {
    static result_type value(parama_type& a, paramb_type& b);
  };
```

де `result_type`, `parama_type`, та `paramb_type` - частини сигнатури `FunctionType`.

4.3 Оцінка Функцій та Вибір Найкращої

Для кожної спеціялізації цього шаблону нам відомі клас кожного із вхідних параметрів. Отже, ми можемо визначити найкращу функцію із наявних ще на етапі компіляції. Розіб'ємо це на два кроки - оцінка усіх функції зі списку за якимось критерієм та вибір елемента із найкращою оцінкою. Для оцінювання типів параметрів функції, із наявних в C++17 інструментів, ми можемо використати `is_same` та `is_base_of`. Для більш детального аналізу відношення між двома класами доведеться зачекати підтримки reflexpr чи реалізовувати користувацьку інспекції генезису. Наприклад, оцінку сумісності двох класів можна визначити як:

```
template<class Parameter, class Argument>
constexpr ssize_t compute_score() noexcept {
  if( std::is_same_v<std::remove_cv_t<Argument>,std::remove_cv_t<Parameter>> ) return
2;
  if( std::is_base_of_v<Parameter, Argument> ) return 1;
  return 0;
}
```

А оцінку для функції як добуток оцінок її параметрів. Цього цілком достатньо для простих ієрархій без множинного успадкування. Для складніших випадків знадобиться дещо складніша функція.

Тепер лишилося тільки заповнити масив такими значеннями для усіх функції із списку:

```
template<auto Entry>
static constexpr function_score make_score(size_t index) noexcept {
  using entry = function_traits<decltype(Entry)>;
  using paramA = typename entry::template nth_arg_type<0>;
  using paramB = typename entry::template nth_arg_type<1>;
  return function_score {
    index, compute_score<paramA,argA>() * compute_score<paramB,argB>() };
}
```

та вибрати елемент із найкращою оцінкою.:

```
template<class ClassA, class ClassB, auto ... Entries>
```

```
constexpr ssize_t find_best_match() noexcept {
  constexpr auto sc = function_scores<ClassA, ClassB, Entries...>{};
  return sc.highest();
}
```

В попередньому фрагменті коду `function_traits` - це допоміжний шаблон для визначення характеристик функції:

```
template<typename Function>
struct function_traits;

template<class Return, class ... Parameter>
struct function_traits<Return (*)(Parameter...)> {
  using result_type = Return;
  static constexpr size_t parameter_count = sizeof...(Parameter);
  template<size_t N>
  using nth_arg_type = std::tuple_element_t<N, std::tuple<Parameter...>>;
};
```

Маючи `constexpr` індекс функції, ми можемо отримати і саму функцію із вхідного списку функцій. Щоб її викликати, знадобиться `down cast`. Оскільки належне відношення типів ми гарантували, можна обійтись `static cast`. Таким чином наш шаблон функції набуде вигляду:

```
template<size_t A, size_t B>
struct func {
  static result_type value(parama_type& a, paramb_type& b) {
    constexpr auto best = find_best_match<type<A>, type<B>, Entries...>();
    if constexpr(best >= 0) {
      constexpr auto f = get_entry<best, Entries...>();
      return (*f)( static_cast<type<A>>(a), static_cast<type<B>>(b));
    } else {
        LOGIC_ERROR("Dispatcher failure");
      }
    }
  }
};
```

Тут у нас виникла потреба в `type<B>` - отримати тип за його ідентифікатором, тобто зворотне мапування. Оскільки ідентифікатори у нас користувацькі, то і зворотне мапування теж може бути тільки користувацьким. Це мапування та постачання максимального ID класу - пов'язані відповідальності, тож можуть бути покладені на один клас. Назвемо його `Domain`. Так як параметрів функції у нас два, і вони можуть бути із різних доменів, то нам потрібні два таких користувацьких домени, а отримувати ми їх будемо через параметри нашого `matrixdispatcher`. Із такими дизайн рішеннями декларація шаблону стає такою:

```
template<typename FunctionType, class DomainA, class DomainB, auto ... Entries>
class matrixdispatch {
public:
  using entry = function_traits<FunctionType>;
  using result_type = typename entry::result_type;
  using parama_type = typename entry::template nth_arg_type<0>;
  using paramb_type = typename entry::template nth_arg_type<1>;
  constexpr matrixdispatch() noexcept {}
private:
  template<size_t A, size_t B>
  struct func {
    static result_type function(parama_type& a, paramb_type& b) {
      //See code snippet above
    }
  };
  static constexpr jumpmatrix<func, DomainA::size, DomainB::size> matrix {};
public:
  static result_type dispatch(parama_type arg1, paramb_type arg2) {
      return matrix[arg1.classID()][arg2.classID()](arg1, arg2);
```

```
  }
};
```

Трохи вдосконаливши шаблон, можна додати підтримку для методів. Що також дозволить використовувати функції поряд із методами. Код для прикладу з цим шаблоном доступний за посиланням: godbolt.org/z/Y89ThK

4.4 Продуктивність

Матричний диспетчинг значно швидший - 48 інструкцій, проте він і значно складніший для компілятора: $O(pnk^2)$ де p кількість параметрів, n - кількість функцій, а k – кількість класів. Для прикладу із 25 класами та 40 функціями час компіляції збільшився на 20 сек.

5 Автоматичне призначення ідентифікаторів

Для допомоги із ідентифікацією класів можна створити такий дизайн класу-домену:

```
template<class ... Classes>
struct domain {
  static constexpr auto size = sizeof...(Classes);
  template<size_t ID>
  using type = std::tuple_element_t<ID, std::tuple<Classes...>>;
  template<class Class>
  static constexpr size_t id_of() noexcept {
    constexpr auto value = index_of<Class, Classes...>();
    return value;
  }
};
```

Цей шаблон отримує перелік класів домену на вхід, реалізує пряме (**id_of**<MyClass>()) та зворотне (type<ID>) мапування.

Щоб дещо спростити компіляцію, можна виключити із матриці абстрактні класи – об'єкти таких класів створити неможливо, отже і в диспетчингу їх ID участі не прийматимуть. Та, оскільки, ID мають бути послідовними, то для ефективного виключення абстрактних класів, їх ID мають зайняти нижній діапазон індексів [0..M]. Щоб вирішити цю проблему у простий спосіб, можна поставити вимогу до списку класів щоб абстрактні класи йшли раніше конкретних і надати механізм валідації цієї вимоги.

6 Порівняння продуктивності

Результати тестів на продуктивність зведені у таблиці нижче. В цій таблиці тест `std::visit` позначає референтну імплементацію з використанням `std::visit` та диспетчингом по аргументам variant-of-object, `std::visit*` - диспетчинг по variant-of-pointers, отриманих через віртуальний виклик методу об'єкту. Стовпчики час містять «настінний» час виконання (wall time) для 10,000,000 викликів над 40 функціями/методами.

	Інструкцій на виклик (ir)		Час (сек)	
	G++-10	CLANG++-11	G++-10	CLANG++-11
`std::visit`	38	25	0.64	1.0
`std::visit*`	58	50	0.93	1.3
`multimethod`	750	702	5.8	6.4
`matrixdispatch`	54	47	0.88	1.2

7 Прикінцеві нотатки

1. У прикладах, викладених на godbolt.org компілятор gcc генерує оптимізований код із статичною диспетчеризацією. Щоб була задіяна, динамічна диспетчеризація слід рознести **test** функції прикладів по різних одиницях компіляції.
2. Приклади із цієї статті також доступні на gist.github.com:
   calculus multidispatcher
   calculus multifunction

shapes multimethod
matrix dispatch

3. Розглянуті рішення також доступні у вигляді include-only бібліотеки:
github.com/hutorny/multimethods

Перелік джерел

[1] Мультиметод. Вікіпедія.
https://uk.wikipedia.org/wiki/Мультиметод

[2] P. Pirkelbauer, Y. Solodkyy, B. Stroustrup,
Open Multi-Methods for C++,
https://www.stroustrup.com/multimethods.pdf

[3] Elsevier B.V.,
EVL: A framework for multi-methods in C++,
https://www.sciencedirect.com/science/article/pii/S0167642314003360

[4] D. Shopyrin,
MultiMethods in C++: Finding a complete solution,
https://www.codeproject.com/Articles/7360/MultiMethods-in-C-Finding-a-complete-solution

[5] Loki Library, Multiple dispatcher,
https://sourceforge.net/projects/loki-lib/ Reference/MutiMethod.h

[6] L. Bettini,
Doublecpp - double dispatch in C++,
http://doublecpp.sourceforge.net/

[7] J. Smith,
Draft proposal for adding Multimethods to C++,
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1529.html

[8] stfairy,
Multiple Dispatch and Double Dispatch,
https://www.codeproject.com/Articles/242749/Multiple-Dispatch-and-Double-Dispatch

[9] J. Smith,
Multimethods,
http://www.op59.net/accu-2003-multimethods.html

[10] D. Shopyrin,
Multimethods in C++ Using Recursive Deferred Dispatching
https://www.computer.org/csdl/magazine/so/2006/03/s3062/13rRUxBa5ve

[11] B. Filipek
How To Use std visit With Multiple Variants
https://www.bfilipek.com/2018/09/visit-variants.html

[12] Wikipedia,
Separation of Concerns
https://en.wikipedia.org/wiki/Separation_of_concerns

[13] Вікіпедія,
Принцип_єдиного_обов'язку,
https://uk.wikipedia.org/wiki/Принцип_єдиного_обов'язку

[14] E. Hutorny,
FNV-1b hash function with extra rotation
https://gist.github.com/hutorny/249c2c67255f842fae08e542f00131b5

[15] A. Mertz
Modern C++ Features – std::variant and std::visit
https://arne-mertz.de/2018/05/modern-c-features-stdvariant-and-stdvisit/