C++

Spring 2021

compscicenter.ru

Грабовой Филипп, t.me/phil-grab

Меркин Николай, t.me/nickolaym

Lection II

Metaprogramming. SFINAE

Метапрограммирование

Суть: программа, порождающая программу, совершающая полезные действия во время компиляции

Картинка про стадии компиляции

pic

Примеры задач и решений

Вычисления на этапе компиляции

- вычисление в compile-time: N-го числа Фибоначчи и т.д.
  - `cout << fib<8>::value; // precomputed in struct fib`
- частичная специализация шаблонов + мемоизация результатов

Контейнеры для разных типов данных

- рантаймовое хранение значений: `std::pair<A, B>` → `std::tuple<A, B, C, …>`
- можно реализовать и compile-time списки/множества и т.д.
  - их элементами могут быть compile-time std::integral_constant
  - + алгоритмы над ними, например `reverse`: `std::tuple<A, B, C>` → `std::tuple<C, B, A>`
- variadic templates + частичная специализация шаблонов

MethodCounters

- есть коллекция, проверить — как дела в `emplace_back` с perfect forwarding
  - perfect == без лишних копирований
- идея: тест с классом, считающим все вызовы нужных методов
- *вопрос*: как написать два, три таких класса без копипасты?

```
template <typename>
struct MethodCounters {
    static inline size_t DefaultConstructed = 0;
    /* ... */
};

#define DefineCountedClass(ClassName)
    struct ClassName {
        using Counters = MethodCounters<ClassName>;
        ClassName() { ++Counters::DefaultConstructed; }
        /* ... */
    };

DefineCountedClass(Class1);
DefineCountedClass(Class2);  // extra, reusing code
```

```
{
    Class1 c1;
    using Counters1 = MethodCounters<Class1>;
    assert(Counters1::DefaultConstructed == 1);

    // pass element to collection
    /* ... emplace_back(std::move(c1)) ... */

    /* assert expected values */
    assert(Counters1::CopyConstructed == 0);
    assert(Counters1::MoveConstructed == 1);
}
```

Serialization

```
struct Person {
    std::string name;
};
struct Building {
    std::string street;
    void visit();
};

void debugLogging(Person& p, Building& b) {
    // как автоматически сгенерировать эти простые метод
    cerr << p.toString();  // 'name: "Alice"'
    cerr << b.toString();  // 'street: "Main Avenue"'
}
```

способ решения: *дописать код*, на этапе компиляции

- *в лоб*: попарсить код регулярками и дописать методы (меняем **source files**)
- *лучше*: через clang frontend работать с AST деревом и дополнять его (меняем **AST**)
- *как еще*: подключить библиотеку с рефлексией, например boost::hana

Interface Style Guide [*]

- В плюсах нет interface — обозначения для pure abstract class без данных и реализованных методов
- Но во многих код-стайлах он есть :) Если оформлять, то просят так:

```
struct SomeInterface {
    // методы: public + pure virtual
    virtual void doSomething() = 0;

    // в классе нет мемберов с данными

    // не забудь!
    virtual ~SomeInterface();
};
```

Interface Style Guide [w/ metaclasses][*]

А можно ли писать меньше + отдать важную работу и проверки компилятору? Например:

```
struct /* какая-то нотация */ SomeInterface {
    // компилятор _сам пометит_
    // все перечисленные методы виртуальными
    void doSomething();

    // компилятор _сам проверит_
    // отсутствие полей-данных в классе

    // компилятор _сам добавит_
    // виртуальный деструктор
};
```

- Такое предлагают в std::meta Metaclass Functions

Другие применения

- реализация единообразного и ad-hoc эффективного интерфейса
  - за счет частичных специализаций: функции над итераторами в STL и др.

Эффекты метапрограммирования

- порождение кода
  - внешние генераторы
  - встроенный препроцессор
  - шаблоны классов, функций, переменных

- преобразования метаданных программы
  - типы
  - синтаксические деревья
  - рефлексия (имена, типы, устройство структур данных) — в STL будет позже
- вычисления времени компиляции
  - constexpr функции и значения

Что дают

- переиспользование кода
- расширение синтаксиса
- эффективные реализации/предподсчеты
- проверки времени компиляции — static_assert, concepts

Метафункции

Функции времени компиляции над метаданными программы (над сведениями об ее элементах)

Встроенные метафункции

- `sizeof`
- `typeid` (*для неполиморфного класса*)
- `decltype`

Шаблоны как метафункции

- у реализации шаблона на входе — аргументы
- на выходе — результат instantiation
  - мемберы — значения времени компиляции
  - результат работы часто значение или тип (один) — *где-то внутри*

```
template <size_t N>
struct fib {
    static const size_t value = ...
};

int main() {
    // входной параметр + результат
    cout << fib<10>::value;
}
```

Приемы метапрограммирования

Рекурсия по множеству аргументов из variadic templates

- Кортеж — хранение всех элементов из списка в одной структуре
- Обобщение `std::pair`, где достаточно `T1 first; T2 second;`

```
template <typename Head, typename... Tail>
struct tuple_element {
    Head                    data;
    tuple_element<Tail...> rest;
}

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

Ветвление

- частичные специализации шаблона класса
- перегрузки шаблона функции про обычным правилам sem01lec10
- + правило SFINAE [*]
- `if constexpr ( /* … */ ) { /* … */ } else { … }`
  - пример применения: `std::advance`, специфический для `RandomAccessIterator`

SFINAE

- SFINAE - Substitution Failure Is Not An Error
- При перегрузки функции ошибочное инстанцирование шаблонов — не ошибка компиляции
  - функция просто выбрасывается из списка кандидатов на наиболее подходящую перегрузку
  - ⇒ нужна "работающая" альтернатива
- SFINAE только про заголовок функции: ошибки в теле будут пропущены

Проверка наличия метода у класса

```
template <class T>
struct is_f_with_strict_signature_defined {
    // might be substitution failure
    template <class Z, void (Z::*)() = &Z::f>
    struct wrapper {};

    template <class C>
    static std::true_type check(wrapper <C> * p);

    template <class C>
    static std::false_type check (...);

    static const bool value = \
        decltype(check<T>(0))::value;
};
```

Возможности STL

- типы-значения: `std::integral_constant`
  - `std::true_type` и `std::false_type`
- SFINAE-селекторы: `std::void_t`, `std::enable_if_t`
- `<type_traits>`
  - изменения типов: `std::add_pointer_t<T>` и др.
  - проверки типов: `std::is_pointer_v<T>`, `std::is_same_v<T1, T2>` ...
- категории итераторов в `<iterator>`

Другие библиотеки

- boost: MetaProgrammingLibrary, fusion, hana
- boost: PreProcessor
- обзор метапрограммирования и возможностей: статья на хабре