Метаклассы в C++17

Фантастика? Реальность!

Сергей Садовников, C++ Russia, Moscow, 20.04.2019

telegram: @flexferrum

GitHub: https://github.com/flexferrum

KASPERSKY lab

Обо мне

Сергей Садовников

Старший архитектор ПО в компании "Лаборатория Касперского", Open Source contributor

e-mail: flexferrum@gmail.com
twitter: @flex_ferrum
telegram: @flexferrum
GitHub: https://github.com/flexferrum

Метаклассы - что это?

- Новая языковая сущность
- Управляет генерацией AST и кода
- “Метапрограммирование” в императивном стиле
- Потенциальная замена макросам и некоторым шаблонам

Метаклассы - что это?

Основное назначение:

- Ещё одна степень обобщения и абстракции: “**interface**”, “**value**”, “**property**”
- “Концепты на стероидах”
- Inplace-расширение компилятора (и языка)

Метаклассы - что это?

1. Основной пропозал: p0707 (rev 3)

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0707r3.pdf

2. Code generation in C++

https://github.com/cor3ntin/CppInjectionReflection/blob/master/doc.md

3. О метаклассах по-русски

https://habr.com/ru/article/448466/

Метаклассы - очень простой пример

```
struct IEnumerator : public IUnknown
{
  virtual ~IEnumerator() {}

  virtual HRESULT Reset() = 0;
  virtual HRESULT MoveNext(BOOL* succeeded) = 0;
  virtual HRESULT Current(IUnknown** item) = 0;
};
```

Метаклассы - очень простой пример

```
struct(interface) IEnumerator
{
  void Reset();
  BOOL MoveNext();
  IUnknown* Current();
};
```

Метаклассы - очень простой пример

```
constexpr void interface(meta::type target, const meta::type source) {
  compiler.require(source.variables().empty(), "interfaces may not contain data");
  for (auto f : source.functions()) {
     compiler.require(!f.is_copy() && !f.is_move(),
                   "interfaces may not copy or move; consider a virtual clone() instead");

     if (!f.has_access())
        f.make_public();

     compiler.require(f.is_public(), "interface functions must be public");
     f.make_pure_virtual();
     f.parameters().add(f.return_type()$ *);
     f.set_type(HRESULT);
     ->(target) f;
  }
  target.bases().push_front(IUnknown);
  ->(target) {virtual ~(source.name()$)() noexcept {}};
};
```

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0707r3.pdf

Метаклассы - очень простой пример

```
constexpr void interface(meta::type target, const meta::type source) {
  compiler.require(source.variables().empty(), "interfaces may not co
  for (auto f : source.functions()) {
      compiler.require(!f.is_copy() && !f.is_move(),
                       "interfaces may not copy or move; consider a virtual clone() instead");

      if (!f.has_access())
          f.make_public();

      compiler.require(f.is_public(), "interface functions must be public");
      f.make_pure_virtual();
      f.parameters().add(f.return_type()$ *);
      f.set_type(HRESULT);
      ->(target) f;
  }
  target.bases().push_front(IUnknown);
  ->(target) {virtual ~(source.name()$)() noexcept {}};
};
```

Декларация-прототип

Метаклассы - очень простой пример

```
constexpr void interface(meta::type target, const meta::type source) {
  compiler.require(source.variables().empty(), "inter
  for (auto f : source.functions()) {
    compiler.require(!f.is_copy() && !f.is_move(),
                     "interfaces may not copy or move; consider a virtual clone() instead");

    if (!f.has_access())
      f.make_public();

    compiler.require(f.is_public(), "interface functions must be public");
    f.make_pure_virtual();
    f.parameters().add(f.return_type()$ *);
    f.set_type(HRESULT);
    ->(target) f;
  }
  target.bases().push_front(IUnknown);
  ->(target) {virtual ~(source.name()$)() noexcept {}};
};
```

Создаваемый инстанс

Метаклассы - очень простой пример

```
constexpr void interface(meta::type target, const meta::type source) {
  compiler.require(source.variables().empty(), "interfaces may not contain data");
  for (auto f : source.functions()) {
    compiler.require(!f.is_copy() && !f.is_move(),
                     "interfaces may not copy or move; consider a virtual clone() instead");

    if (!f.has_access())
      f.make_public();

    compiler.require(f.is_public(), "interface functions must be public");
    f.make_pure_virtual();
    f.parameters().add(f.return_type()$ *);
    f.set_type(HRESULT);
    ->(target) f;
  }
  target.bases().push_front(IUnknown);
  ->(target) {virtual ~(source.name()$)() noexcept {}};
};
```

Проверка свойств

Метаклассы - очень простой пример

```
constexpr void interface(meta::type target, const meta::type source) {
  compiler.require(source.variables().empty(), "interfaces may not contain data");
  for (auto f : source.functions()) {
     compiler.require(!f.is_copy() && !f.is_move(),
                 "interfaces may not copy or move; consider a virtual clone() instead");

     if (!f.has_access())
        f.make_public();

     compiler.require(f.is_public(), "interface functions must be public");
     f.make_pure_virtual();
     f.parameters().add(f.return_type()$ *);
     f.set_type(HRESULT);
     ->(target) f;
  }
  target.bases().push_front(IUnknown);
  ->(target) {virtual ~(source.name()$)() noexcept {}};
};
```

Модификация свойств

Метаклассы - очень простой пример

```
constexpr void interface(meta::type target, const meta::type source) {
  compiler.require(source.variables().empty(), "interfaces may not contain data");
  for (auto f : source.functions()) {
      compiler.require(!f.is_copy() && !f.is_move(),
                       "interfaces may not copy or move; consider a virtual clone() instead");

      if (!f.has_access())
          f.make_public();

      compiler.require(f.is_public(), "interface functions must be public");
      f.make_pure_virtual();
      f.parameters().add(f.return_type()$ *);
      f.set_type(HRESULT);
      ->(target) f;
  }
  target.bases().push_front(IUnknown);
  ->(target) {virtual ~(source.name()$)() noexcept {}};
};
```

Code injection

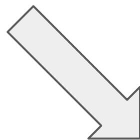
Метаклассы - очень простой пример

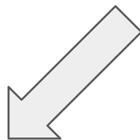
```
struct(interface) IEnumerator
{
  void Reset();
  BOOL MoveNext();
  IUnknown* Current();
};
```

```
constexpr void interface(meta::type target, const meta::type source) {
  compiler.require(source.variables().empty(), "interfaces may not contain data");
  for (auto f : source.functions()) {
    compiler.require(!f.is_copy() && !f.is_move(),
                 "interfaces may not copy or move; consider a virtual clone()
instead");

    if (!f.has_access())
      f.make_public();

    compiler.require(f.is_public(), "interface functions must be public");
    f.make_pure_virtual();
    f.parameters().add(f.return_type()$ *);
    f.set_type(HRESULT);
    ->(target) f;
  }
  target.bases().push_front(IUnknown);
  ->(target) {virtual ~(source.name()$)() noexcept {}};
};
```





```
struct IEnumerator : public IUnknown
{
  virtual ~IEnumerator() {}

  virtual HRESULT Reset() = 0;
  virtual HRESULT MoveNext(BOOL* succeeded) = 0;
  virtual HRESULT Current(IUnknown** item) = 0;
};
```

Метаклассы - проблемы

- Базируется на ещё не принятых пропозалах -> предлагаемый синтаксис может меняться
- Неизвестно, когда всё это примут. По некоторым оценкам - не раньше C++2z
- Всё ещё развивается и дорабатывается -> что получится в итоге - не очень ясно

Метаклассы - а если хочется раньше?

- Ждать принятия в стандарт и поддержки компиляторами
- Использовать шаблонную магию и макросы
- Реализовать и использовать сторонние утилиты

Метаклассы - а если хочется раньше?

- Ждать принятия в стандарт и поддержки компиляторами
- Использовать шаблонную магию и макросы
- **Реализовать и использовать сторонние утилиты**

Метаклассы - сторонняя утилита

Достоинства:

- Не требуется особой поддержки со стороны компиляторов
- Реализация в рамках актуальных стандартов
- Может быть легко встроена в сборочный тулчейн
- Может сделать кодовую базу проектов проще и упростить поддержку

Метаклассы - сторонняя утилита

Недостатки:

- Не соответствует одной из целей создания метаклассов: “Eliminate the need to invent non-C++ “side languages” and special compilers, such as Qt moc, COM MIDL, and C++/CX”
- Собственный “птичий” язык для описания специальных конструкций
- Не всегда удобное использование предлагаемых возможностей
- Потенциальная несовместимость с итоговым вариантом пропозала

Метаклассы - сторонняя утилита

Основные возможности:

- Позволяет объявлять метаклассы
- Позволяет объявлять прототипы конкретных метаклассов
- В метаклассах позволяет проверять свойства прототипов и генерировать диагностику
- Позволяет внедрять в инстансы метаклассов код и сущности

Metaclasses processing tool - пример

```
template<typename ... Types>
inline void CRTPVisitor(meta::ClassInfo dst, const meta::ClassInfo& src)
{
  const auto& tplParam = dst.add_template_type_param("Derived");

  for (auto& f : src.functions())
     $_inject_v(dst, public) f;

  $_inject_v(dst, public) [name="operator()"](auto&& obj) -> void {
     $_v("GetDerived")()->$_mem("Visit")(std::forward<$_str(T0)>(obj));
  };

  for (auto& t : t_$(Types ...))
     $_inject_v(dst, public) [name="Visit"](const $_t(t)& obj) -> void {};

  $_inject_v(dst, private) [name="GetDerived"]() -> $_t(tplParam)* {
     return static_cast<$_t(tplParam)*>($_str(this));
  };
}
```

Metaclasses processing tool - пример

```
template<typename ... Types>
inline void CRTPVisitor(meta::ClassInfo dst, const meta::ClassInfo& src)
{
  const auto& tplParam = dst.add_template_type_param("Derived");

  for (auto& f : src.functions())
     $_inject_v(dst, public) f;

  $_inject_v(dst, public) [name="operator()"](auto&& obj) -> void {
     $_v("GetDerived")()->$_mem("Visit")(std::forward<$_str(T0)>(obj));
  };

  for (auto& t : t_$(Types ...))
     $_inject_v(dst, public) [name="Visit"](const $_t(t)& obj) -> void {};

  $_inject_v(dst, private) [name="GetDerived"]() -> $_t(tplParam)* {
     return static_cast<$_t(tplParam)*>($_str(this));
  };
}
```

Добавляем шаблонный параметр в инстанс

Metaclasses processing tool - пример

```
template<typename ... Types>
inline void CRTPVisitor(meta::ClassInfo dst, const meta::ClassInfo& src)
{
  const auto& tplParam = dst.add_template_type_param("Derived");

  for (auto& f : src.functions())
      $_inject_v(dst, public) f;

  $_inject_v(dst, public) [name="operator()"](auto&& obj) -> void {
    $_v("GetDerived")()->$_mem("Visit")(std::forward<$_str(T0)>(obj));
  };

  for (auto& t : t_$(Types ...))
    $_inject_v(dst, public) [name="Visit"](const $_t(t)& obj) -> void {};

  $_inject_v(dst, private) [name="GetDerived"]() -> $_t(tplParam)* {
    return static_cast<$_t(tplParam)*>($_str(this));
  };
}
```

Копируем функции из исходного объявления в инстанс

Metaclasses processing tool - пример

```
template<typename ... Types>
inline void CRTPVisitor(meta::ClassInfo dst, const meta::ClassInfo& src)
{
  const auto& tplParam = dst.add_template_type_param("Derived");

  for (auto& f : src.functions())
    $_inject_v(dst, public) f;

  $_inject_v(dst, public) [name="operator()"](auto&& obj) -> void {
      $_v("GetDerived")()->$_mem("Visit")(std::forward<$_str(T0)>(obj));
  };

  for (auto& t : t_$(Types ...))
    $_inject_v(dst, public) [name="Visit"](const $_t(t)& obj) -> void {};

  $_inject_v(dst, private) [name="GetDerived"]() -> $_t(tplParam)* {
    return static_cast<$_t(tplParam)*>($_str(this));
  };
}
```

Добавляем оператор вызова функции

Metaclasses processing tool - пример

```
template<typename ... Types>
inline void CRTPVisitor(meta::ClassInfo dst, const meta::ClassInfo& src)
{
  const auto& tplParam = dst.add_template_type_param("Derived");

  for (auto& f : src.functions())
    $_inject_v(dst, public) f;

  $_inject_v(dst, public) [name="operator()"](auto&& obj) -> void {
    $_v("GetDerived")()->$_mem("Visit")(std::forward<$_str(T0)>(obj));
  };

  for (auto& t : t_$(Types ...))
     $_inject_v(dst, public) [name="Visit"](const $_t(t)& obj) -> void {};

  $_inject_v(dst, private) [name="GetDerived"]() -> $_t(tplParam)* {
    return static_cast<$_t(tplParam)*>($_str(this));
  };
}
```

Добавляем fallback-методы для всех типов из Types

Metaclasses processing tool - пример

```
template<typename ... Types>
inline void CRTPVisitor(meta::ClassInfo dst, const meta::ClassInfo& src)
{
  const auto& tplParam = dst.add_template_type_param("Derived");

  for (auto& f : src.functions())
     $_inject_v(dst, public) f;

  $_inject_v(dst, public) [name="operator()"](auto&& obj) -> void {
     $_v("GetDerived")()->$_mem("Visit")(std::forward<$_str(T0)>(obj));
  };

  for (auto& t : t_$(Types ...))
     $_inject_v(dst, public) [name="Visit"](const $_t(t)& obj) -> void {};

  $_inject_v(dst, private) [name="GetDerived"]() -> $_t(tplParam)* {
     return static_cast<$_t(tplParam)*>($_str(this));
  };
}
```

Добавляем метод GetDrived()

Metaclasses processing tool - пример

```
$_class(SomeVisitor, CRTPVisitor<A, B, C>)
{
public:
   void TestMethod1();
   std::string TestMethod2(int param) const;
};
```

Metaclasses processing tool - пример

```
$_class(SomeVisitor, CRTPVisitor<A, B, C>)
{
public:
   void TestMethod1();
   std::string TestMethod2(int param) const;
};
```

Название будущего инстанса метакласса

Metaclasses processing tool - пример

```
$_class(SomeVisitor, CRTPVisitor<A, B, C>)
{
public:
   void TestMethod1();
   std::string TestMethod2(int param) const;
};
```

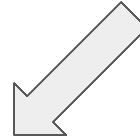
Metaclasses processing tool - пример

```
$_class(SomeVisitor, CRTPVisitor<A, B, C>)
{
public:
  void TestMethod1();
  std::string TestMethod2(int param)
const;
};
```

Тело прототипа

Metaclasses processing tool - пример

```
$_class(SomeVisitor, CRTPVisitor<A, B, C>)
{
public:
   void TestMethod1();
   std::string TestMethod2(int param) const;
};
```



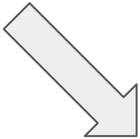
```
template<typename ... Types>
inline void CRTPVisitor(meta::ClassInfo dst, const meta::ClassInfo& src)
{
  const auto& tplParam = dst.add_template_type_param("Derived");

  for (auto& f : src.functions())
     $_inject_v(dst, public) f;

  $_inject_v(dst, public) [name="operator()"](auto&& obj) -> void {
     $_v("GetDerived")()->$_mem("Visit")(std::forward<$_str(T0)>(obj));
  };

  for (auto& t : t_$(Types ...))
     $_inject_v(dst, public) [name="Visit"](const $_t(t)& obj) -> void {};

  $_inject_v(dst, private) [name="GetDerived"]() -> $_t(tplParam)* {
     return static_cast<$_t(tplParam)*>($_str(this));
  };
}
```



```
template <typename Derived> class SomeVisitor {
public:
 void TestMethod1();
 std::string TestMethod2(int param) const;
 template <typename T0> void operator()(T0 &&obj) {
   GetDerived()->Visit(std::forward<T0>(obj));
 }
 void Visit(const A &obj) {}
 void Visit(const B &obj) {}
 void Visit(const C &obj) {}

protected:
private:
 Derived *GetDerived() { static_cast<Derived *>(this); }
};
```

Metaclasses processing tool - пример

```
template <typename Derived> class SomeVisitor {
public:
  void TestMethod1();
  std::string TestMethod2(int param) const;
  template <typename T0> void operator()(T0 &&obj) {
     GetDerived()->Visit(std::forward<T0>(obj));
  }
  void Visit(const A &obj) {}
  void Visit(const B &obj) {}
  void Visit(const C &obj) {}

protected:
private:
  Derived *GetDerived() { static_cast<Derived *>(this); }
};
```

Metaclasses processing tool - пример

```
template <typename Derived> class SomeVisitor {
public:
  void TestMethod1();
  std::string TestMethod2(int param) const;
  template <typename T0> void operator()(T0 &&obj) {
     GetDerived()->Visit(std::forward<T0>(obj));
  }
  void Visit(const A &obj) {}
  void Visit(const B &obj) {}
  void Visit(const C &obj) {}

protected:
private:
  Derived *GetDerived() { static_cast<Derived *>(this); }
};
```

Добавленный шаблонный параметр

Metaclasses processing tool - пример

```
template <typename Derived> class SomeVisitor {
public:
  void TestMethod1();
  std::string TestMethod2(int param) const;
  template <typename T0> void operator()(T0 &&obj) {
    GetDerived()->Visit(std::forward<T0>(obj));
  }
  void Visit(const A &obj) {}
  void Visit(const B &obj) {}
  void Visit(const C &obj) {}

protected:
private:
  Derived *GetDerived() { static_cast<Derived *>(this); }
};
```

Объявления из декларации

Metaclasses processing tool - пример

```
template <typename Derived> class SomeVisitor {
public:
  void TestMethod1();
  std::string TestMethod2(int param) const;
  template <typename T0> void operator()(T0 &&obj) {
      GetDerived()->Visit(std::forward<T0>(obj));
   }
  void Visit(const A &obj) {}
  void Visit(const B &obj) {}
  void Visit(const C &obj) {}

protected:
private:
  Derived *GetDerived() { static_cast<Derived *>(this); }
};
```

Оператор вызова функции

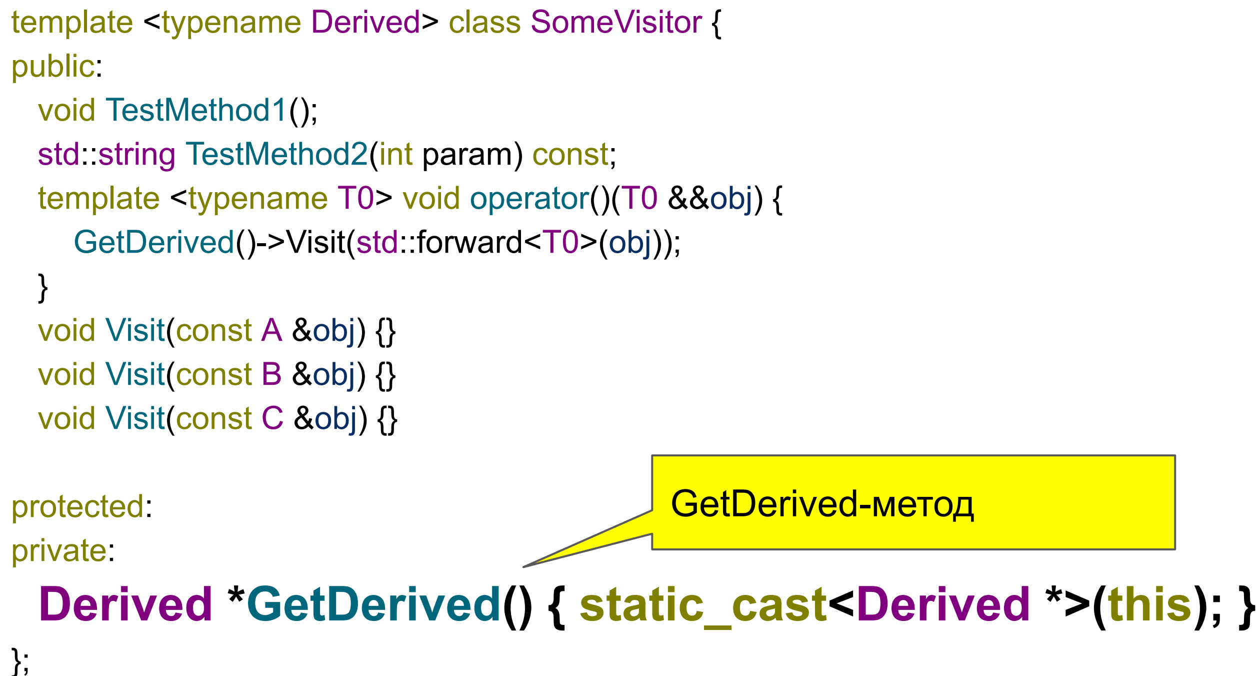
Metaclasses processing tool - пример

```
template <typename Derived> class SomeVisitor {
public:
   void TestMethod1();
   std::string TestMethod2(int param) const;
   template <typename T0> void operator()(T0 &&obj) {
      GetDerived()->Visit(std::forward<T0>(obj));
   }
   void Visit(const A &obj) {}
   void Visit(const B &obj) {}
   void Visit(const C &obj) {}

protected:
private:
   Derived *GetDerived() { static_cast<Derived *>(this); }
};
```

Fallback-методы

Metaclasses processing tool - пример




```
template <typename Derived> class SomeVisitor {
public:
   void TestMethod1();
   std::string TestMethod2(int param) const;
   template <typename T0> void operator()(T0 &&obj) {
      GetDerived()->Visit(std::forward<T0>(obj));
   }
   void Visit(const A &obj) {}
   void Visit(const B &obj) {}
   void Visit(const C &obj) {}

protected:
private:
   Derived *GetDerived() { static_cast<Derived *>(this); }
};
```

Metaclasses processing tool - особенности

- Отдельная от компилятора утилита
- Эмулирует (а не полноценно реализует) code injection
- Использует (и будет использовать) ‘$’-синтаксис
- Генерирует исходные файлы с новыми декларациями, а не модифицирует AST
- (как следствие) Доступна отладка сгенерированного кода
- API базируется на стандарте C++14
- (в будущем) реализует static reflection на основе [TS N4766](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4766.pdf)
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4766.pdf

Metaclasses processing tool - фронтенд

- Комплект макросов и для написания метаклассов и их прототипов (**$class**, **$constexpr** и т. п.)
- Декларации static reflection API (**ClassInfo**, **TypeInfo** и т. п.)
- API специального назначения (**meta::compiler, meta::project** и т. п.)

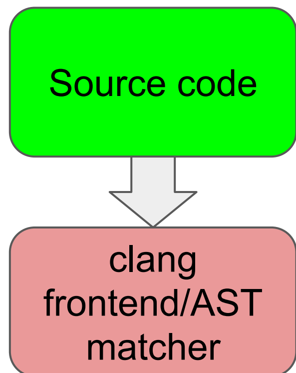
Metaclasses processing tool - бэкэнд

- Механизмы поиска метаклассов и их прототипов в коде
- Реализация C++-интерпретатора для исполнения тела метакласса
- Реализация механизмов code injection
- Реализация static reflection и прочих элементов API
- Генерация финального кода для инстансов метаклассов

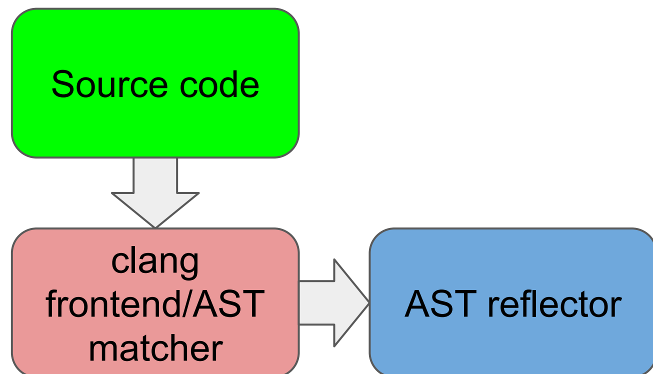
Metaclasses processing tool - под капотом

Source code

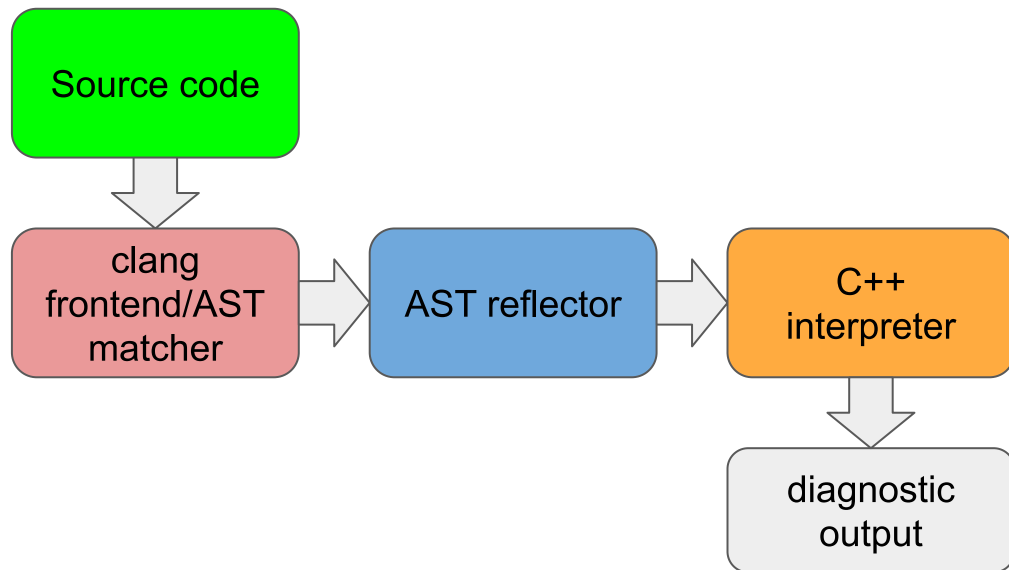
Metaclasses processing tool - под капотом



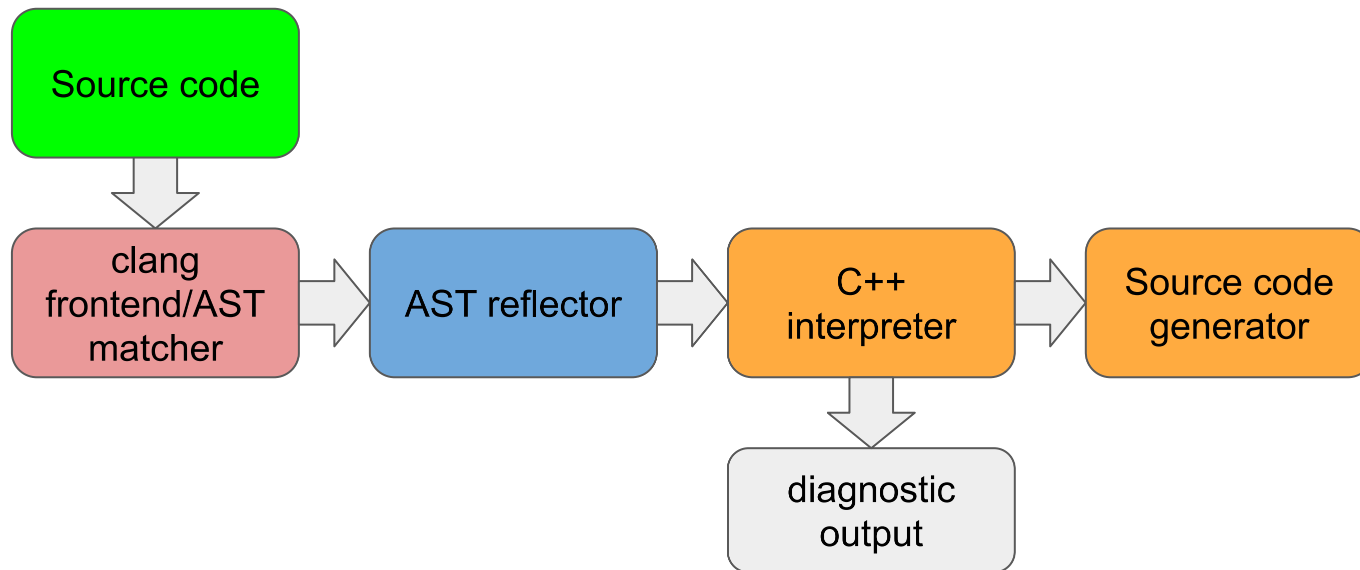
Metaclasses processing tool - под капотом



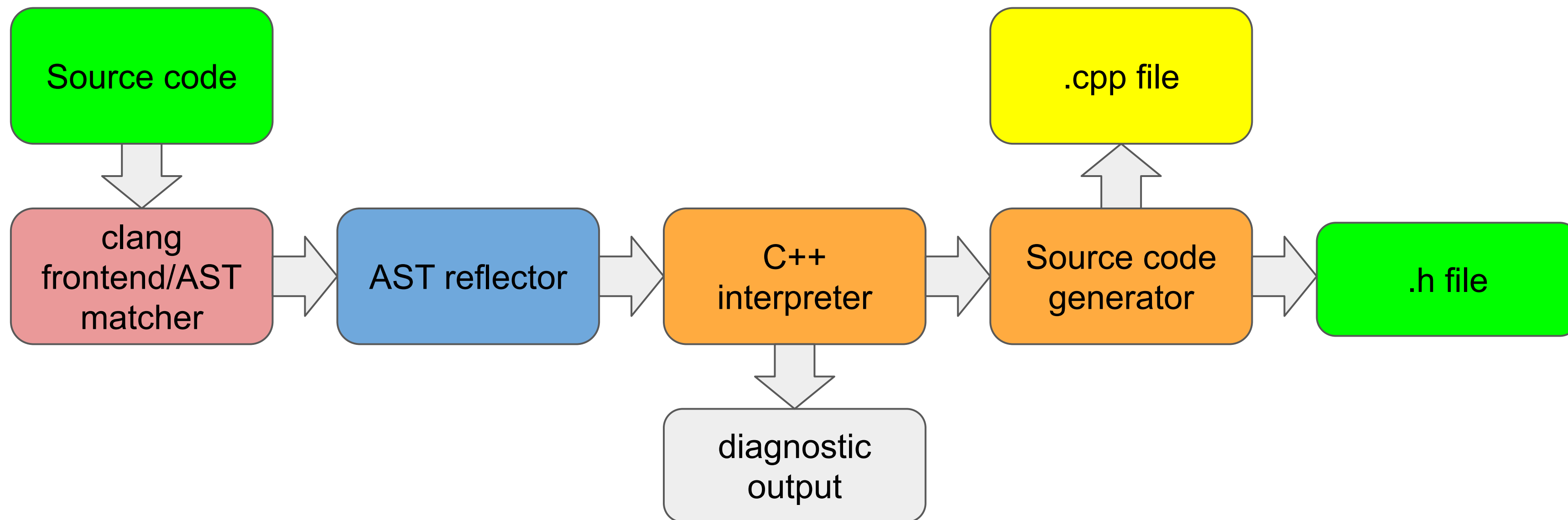
Metaclasses processing tool - под капотом



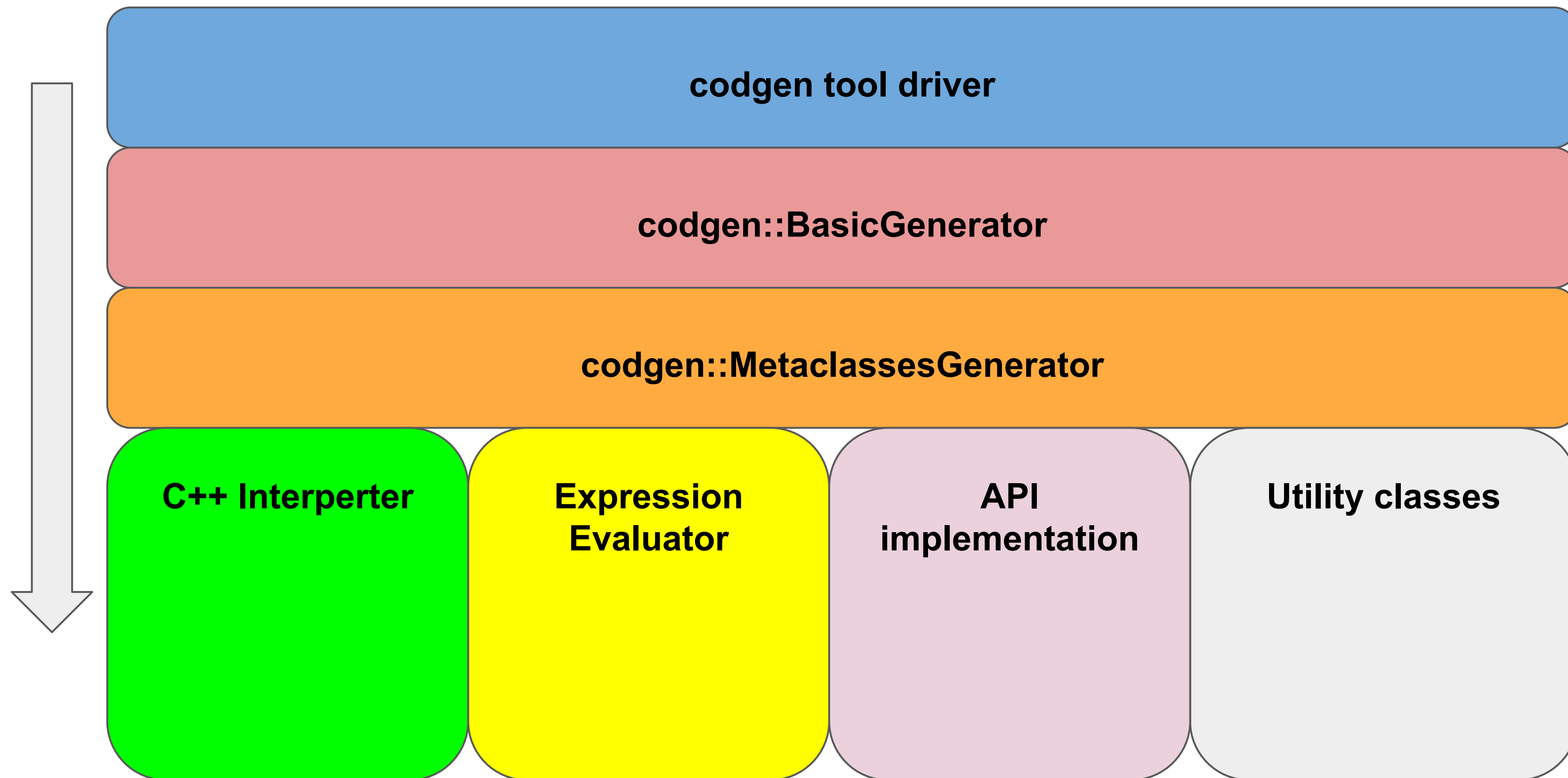
Metaclasses processing tool - под капотом



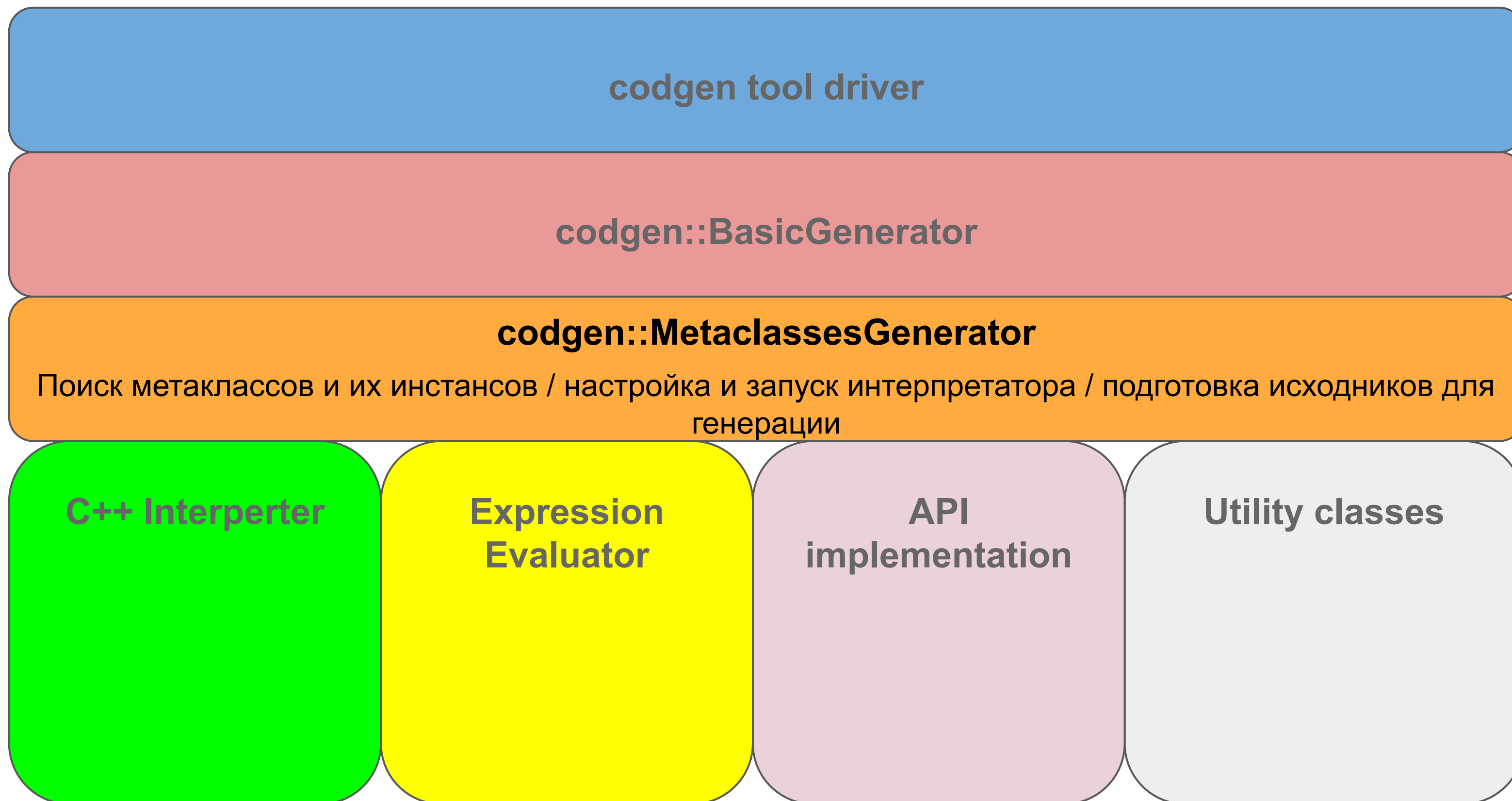
Metaclasses processing tool - под капотом



Metaclasses processing tool - под капотом



Поиск прототипов метаклассов



Поиск прототипов метаклассов

```
$_class(SomeVisitor, CRTPVisitor<A, B, C>)
{
public:
   void TestMethod1();
   std::string TestMethod2(int param) const;
};
```

Поиск прототипов метаклассов

```
struct MetaClassInstance_SomeVisitor : public meta::detail::MetaClassImplBase
{
  static constexpr std::initializer_list<meta::MetaclassMethodPtr> metaPtrList_ = {CRTPVisitor<A, B, C>};
  class SomeVisitor;
};

class MetaClassInstance_SomeVisitor::SomeVisitor {
public:
  void TestMethod1();
  std::string TestMethod2(int param) const;
};
```

Поиск прототипов метаклассов

```
struct MetaClassInstance_SomeVisitor : public meta::detail::MetaClassImplBase
{
    static con                                   rList_ = {CRTPVisitor<A, B, C>};
    class So
};

class MetaClassInstance_SomeVisitor::SomeVisitor {
public:
    void TestMethod1();
    std::string TestMethod2(int param) const;
};
```

```
cxxRecordDecl(isDerivedFrom("meta::detail::MetaClassImplBase"))
```

Поиск прототипов метаклассов

```
struct MetaClassInstance_SomeVisitor : public meta::detail::MetaClassImplBase
{
  static constexpr std::initializer_list<meta::MetaclassMethodPtr> metaPtrList_ = {CRTPVisitor<A, B, C>};
  class SomeVisitor;
};

class MetaClassInstance_SomeVisitor::SomeVisitor {
public:
  void TestMethod1();
  std::string TestMethod2(int param) const;
};
```

Список функций-метаклассов

Поиск прототипов метаклассов

```
struct MetaClassInstance_SomeVisitor : public meta::detail::MetaClassImplBase
{
  static constexpr std::initializer_list<meta::MetaclassMethodPtr> metaPtrList_ = {CRTPVisitor<A, B, C>};
  class SomeVisitor;
};
```

```
class MetaClassInstance_SomeVisitor::SomeVisitor {
public:
  void TestMethod1();
  std::string TestMethod2(int param) const;
};
```

Объявление прототипа инстанса

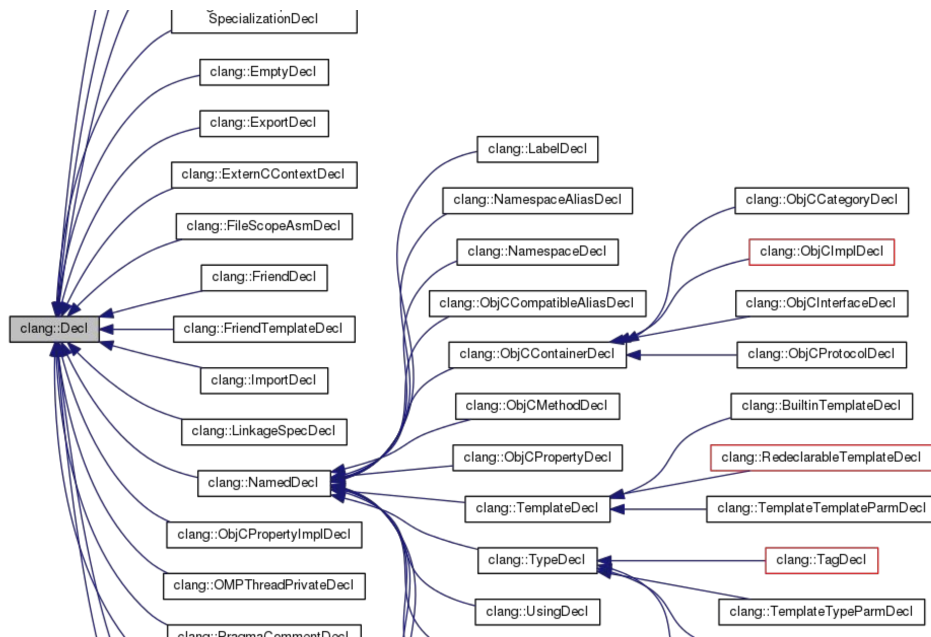
Работа MetaclassesGenerator

1. clang AST matcher вызывает callback на каждый встреченный класс-наследник **meta::detail::MetaClassImplBase**
2. Анализируется тело объявления класса
3. Тело прототипа пропускается через AST Reflector
4. Для каждого метакласса вызывается интерпретатор
5. По результату работы интерпретатора генерируется итоговый исходный код

AST Reflector - зачем?

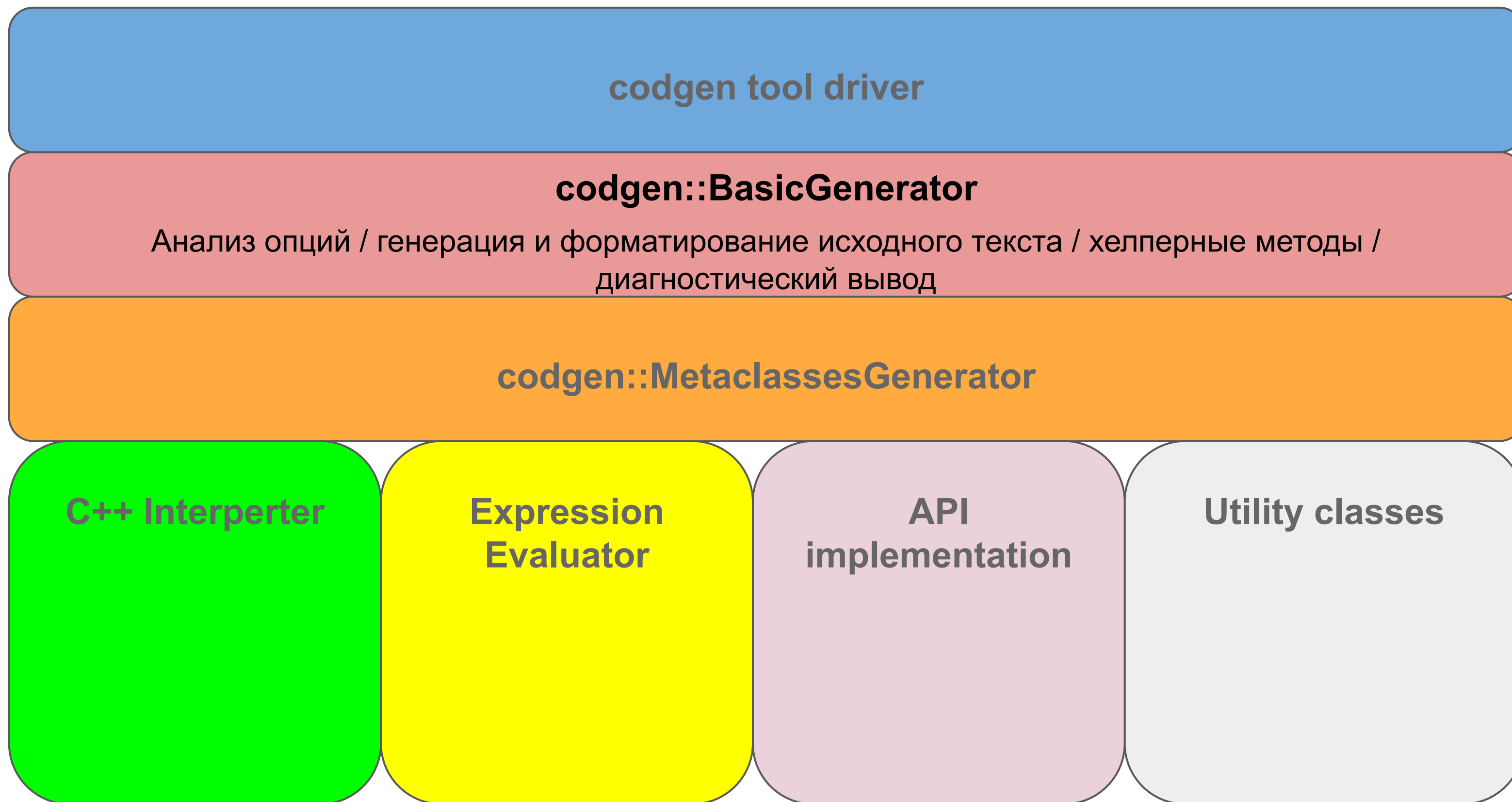
- clang AST крайне сложна и детализирована
- более просто отобразить на reflection API
- проще анализировать
- просто отобразить внутрь текстового шаблонизатора

AST Reflector - зачем?



reflection::TypeInfo
reflection::MemberInfo
reflection::MethodInfo
reflection::ClassInfo
reflection::EnumInfo
reflection::NamespaceInfo

Генерация кода



Генерация кода

1. Выполняется только если исполнение метаклассов прошло без ошибок
2. Результат отображается в движок текстовых шаблонов на базе Jinja2
3. Грузится шаблон результирующего файла
4. По загруженному шаблону формируется финальный текст
5. Полученный текст форматируется с помощью clang format и сохраняется на диск

Генерация кода - шаблон

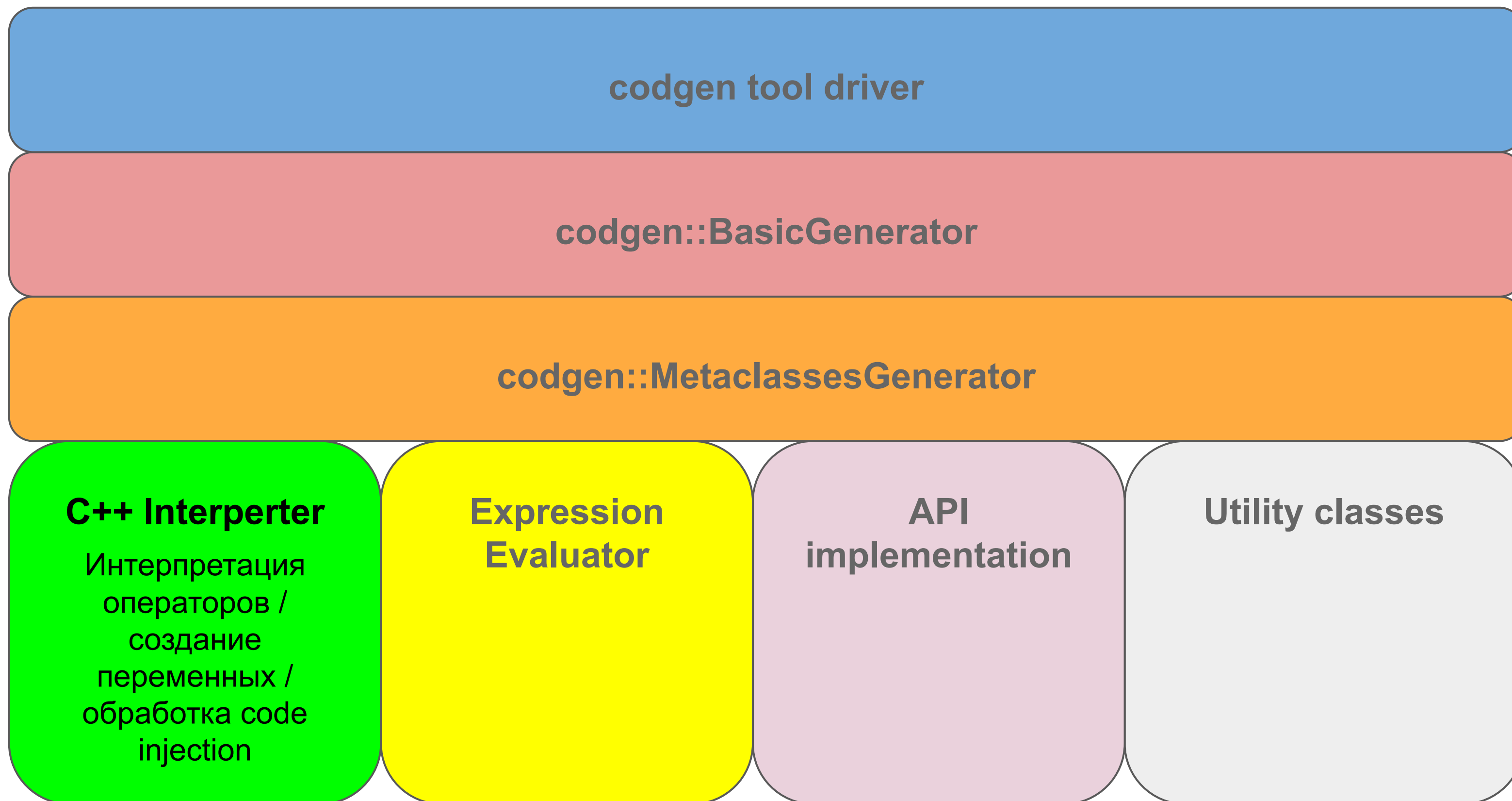
```
{% for class in classes | sort(attribute="name") %}
{% macro Decls(class, access) %}

{% for member in class.members | selectattr('accessType', 'in', access) %}
  {{'static ' if member.isStatic}}{{member.type.printedName}} {{ member.name }};
{% endfor %}
{% endmacro %}

{% if class.isTemplate %}template< {{class.tplParams | map(attribute='tplDeclName') | join(', ') }} >{% endif %}
class {{ class.name }}
{
public:
  {{ Decls(class, ['Public']) }}
protected:
  {{ Decls(class, ['Protected']) }}
private:
  {{ Decls(class, ['Private', 'Undefined']) }}
};

{% endfor %}
{% endblock %}
```

Интерпретатор



Интерпретатор

1. Реализован на основе switch/case по типу узла AST
2. Детально анализирует AST функции-метакласса
3. Фактически исполняет код метакласса оператор за оператором
4. Создаёт инстанс метакласса в соответствии с прописанной логикой посредством инжекта в него фрагментов кода и сущностей
5. Вызывает в процессе интерпретации intrinsic-методы

Интерпретатор

```
for (auto& t : t_$(Types ...))
{
   $_inject_v(dst, public) [name="Visit"](const $_t(t)& obj) -> void
   {
      $_v("GetDerived")()->$_mem("Visit")(std::forward<$_str(T0)>(obj));
   };
}
```

Интерпретатор

```
for (auto& t : meta::reflect_type<Types ...>())
{
  [[gsl::suppress("inject", "$target=dst", "public")]]
  [name="Visit"](const decltype(meta::project_type(t))& obj) -> void
  {
     meta::project("GetDerived")()->
           $_project_member("Visit")(std::forward<meta::project("T0")>(obj));
  };
}
```

Интерпретатор

```
for (auto& t : meta::reflect_type<Types ...>())
{
  [[gsl::suppress("inject", "$target=dst", [illegible]")]]
  [name="Visit"](const decltype(meta::project_type(t))& obj) -> void
  {
     meta::project("GetDerived")()->
            $_project_member("Visit")(std::forward<meta::project("T0")>(obj));
  };
}
```

intrinsic-методы

KASPERSKY lab

Интерпретатор

```
for (auto& t : meta::reflect_type<Types ...>())
{
  [[gsl::suppress("inject", "$target=dst", "public")]]
  [name="Visit"](const decltype(meta::project_type(t))& obj) -> void
  {
    meta::project("GetDerived")()->
        $_project_member("Visit")(std::forward<meta::project("T0")>(obj));
  };
}
```

Атрибут, отмечающий code injection

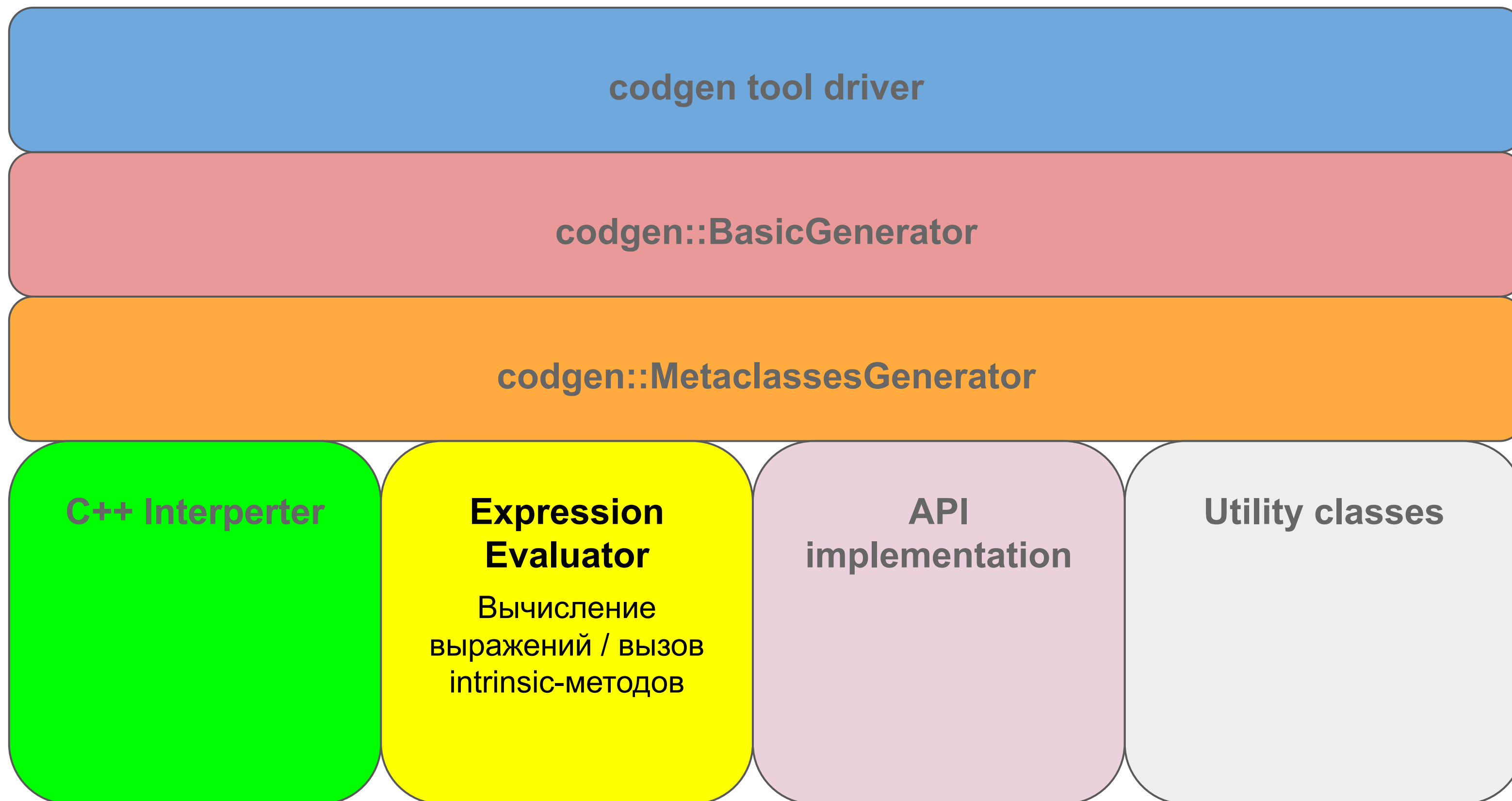
Интерпретатор

- Операторы исполняются в соответствии со своей семантикой
- Выражения вычисляются в соответствии с построенным AST
- Особая обработка intrinsic-методов в зависимости от контекста
- Операторы, помеченные специальным атрибутом, интерпретируются в зависимости от контекста
- Декларации переменных создают переменные
- Помеченное для инжекта в зависимости от своего вида добавляется как:
    - Декларация
    - Фрагмент кода
    - Реализация метода

Интерпретатор - отличия от constexpr

- Более тесная интеграция с инфраструктурой процессора метаклассов
- Нет необходимости в точном представлении вычисляемых типов
- Результат - новая декларация, а не constexpr-значение
- Другой набор ограничений на используемые конструкции
- Наличие code injection

Вычислитель выражений



Вычислитель выражений

- Реализован на базе **clang::ConstStmtVisitor<...>**
- Корректность выражения с точки зрения типов гарантируется clang frontend
- Вычислитель использует универсальный value-тип на базе boost::variant
- Именно здесь обрабатываются intrinsic-методы
- При необходимости идёт обращение к сущностям отрефлексированной AST

Вычислитель выражений - Value

Value:

- Empty
- Void
- Bool
- Int
- Float
- String
- RelfectedObject
- Reference
- Pointer

ReflectedObject:

- Compiler
- ClassInfo
- MethodInfo
- MemberInfo
- TypeInfo
- RangeT
- IteratorT

Вычислитель выражений - intrinsic-методы

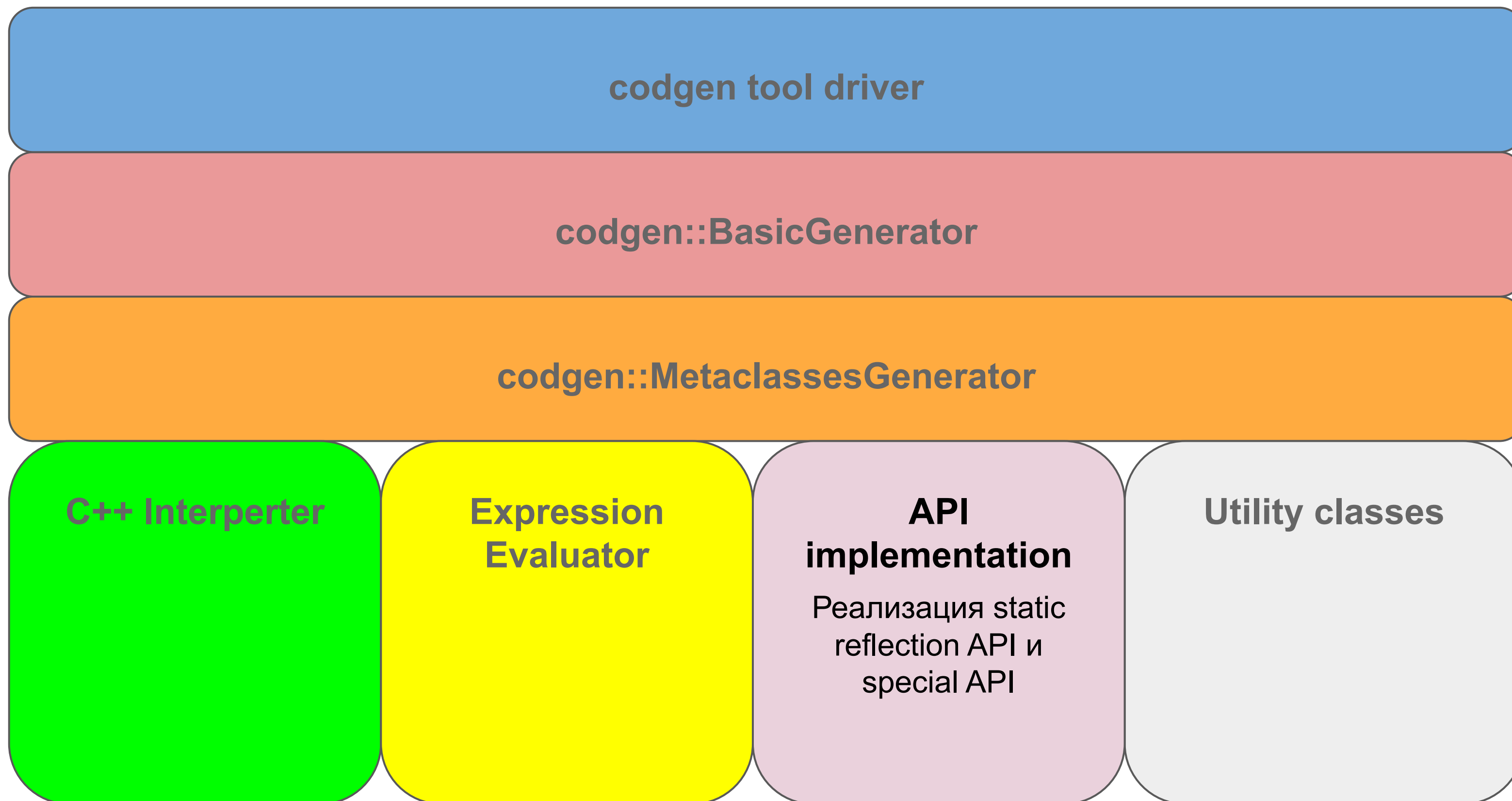
```
void ExpressionEvaluator::VisitCallExpr(const CallExpr* expr) {
  ...
  std::string fnName = expr->getDirectCallee()->getQualifiedNameAsString();

  Value result;
  if (fnName == "meta::reflect_type") {
    const clang::TemplateArgumentList* tplArgs = function->getTemplateSpecializationArgs();
    if (tplArgs == nullptr) {
      return;
    }

    std::vector<reflection::TypeInfoPtr> types;
    ExtractTemplateTypeArgs(tplArgs->asArray(), types, m_interpreter->m_astContext);

    auto range = MakeStdCollectionRefRange(std::move(types));
    result = Value(ReflectedObject(range));
    m_evalResult = true;
    vScope.Submit(std::move(result));
  }
... }
```

Вычислитель выражений - static reflection



Вычислитель выражений - static reflection

```
bool ReflectedMethods::ClassInfo_variables(InterpreterImpl* interpreter, reflection::ClassInfoPtr obj, Value& result)
{
   auto rangePtr = MakeStdCollectionRefRange(obj->members);
   result = Value(ReflectedObject(rangePtr));

   return true;
}

bool ReflectedMethods::ClassInfo_functions(InterpreterImpl* interpreter, reflection::ClassInfoPtr obj, Value& result)
{
   auto methods = obj->methods;
   auto rangePtr = MakeStdCollectionRefRange(std::move(methods));
   result = Value(ReflectedObject(rangePtr));

   return true;
}
```

Вычислитель выражений - static reflection

```
bool CallMember(InterpreterImpl* interpreter, Value& obj, const clang::CXXMethodDecl* method, const std::vector<Value>& args,
Value& result) {
        static std::unordered_map<std::string, ThunkInvoker> fns = {
        {"meta::CompilerImpl::message/void message(const char *msg)"s, thunkMaker(&ReflectedMethods::Compiler_message)},
        {"meta::CompilerImpl::require/void require(bool, const char *message)"s, thunkMaker(&ReflectedMethods::Compiler_require)},
        {"meta::ClassInfo::variables/Range<meta::MemberInfo> &variables() const"s,
thunkMaker(&ReflectedMethods::ClassInfo_variables)},
        {"meta::ClassInfo::functions/Range<meta::MethodInfo> &functions() const"s,
thunkMaker(&ReflectedMethods::ClassInfo_functions)},
        {"meta::ClassInfo::add/template<> void add<meta::MethodInfo>(meta::MethodInfo entity, meta::AccessType access =
AccessType::Unspecified)"s, thunkMaker(&ReflectedMethods::ClassInfo_addMethod)},
        ...
    };

    auto p = fns.find(methodName);

    return p->second(thunk, interpreter, obj, args, result);
}
```

Вычислитель выражений - static reflection

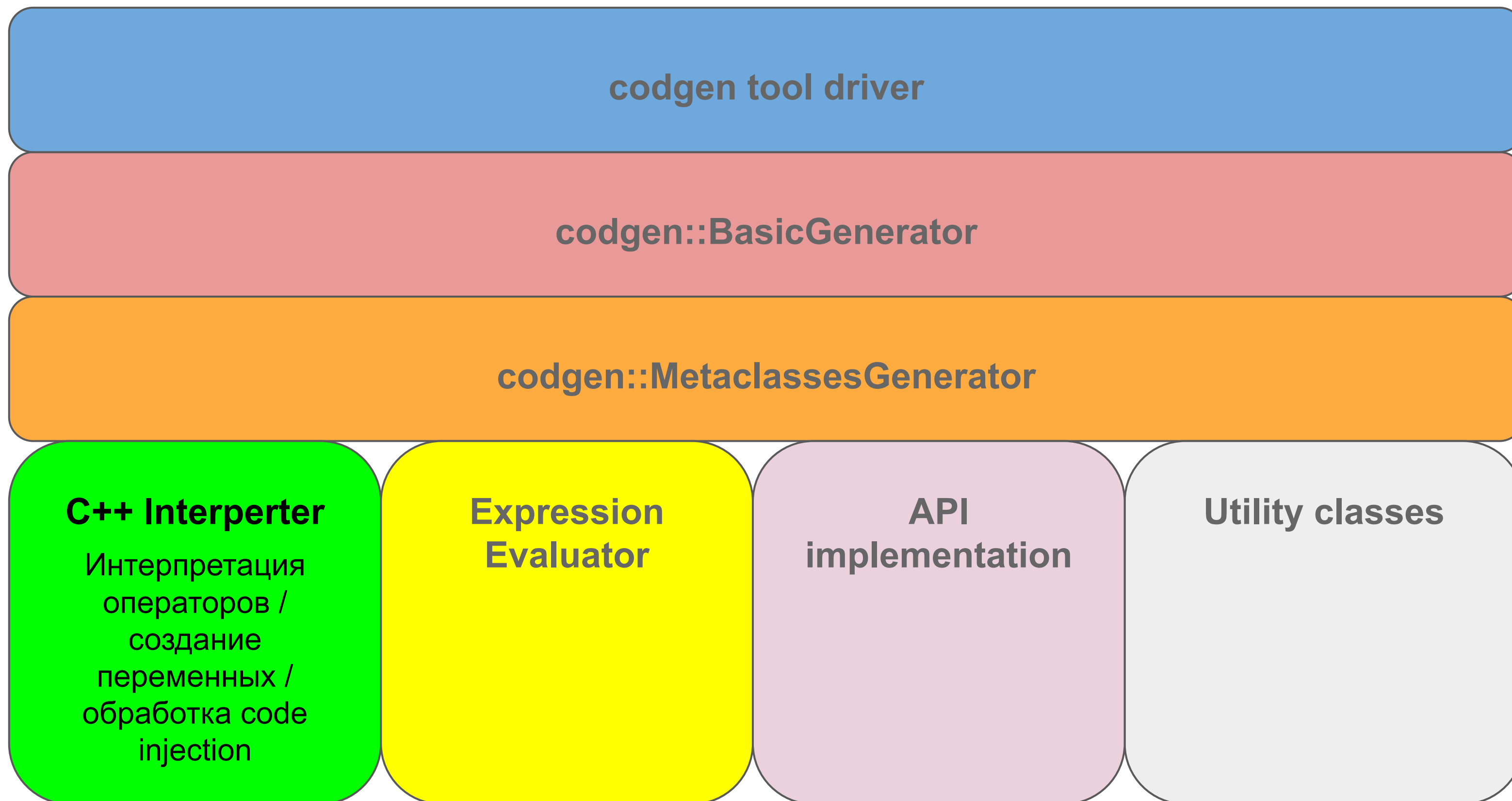
```
void ExpressionEvaluator::VisitCXXMemberCallExpr(const clang::CXXMemberCallExpr* expr)
{
  VisitorScope vScope(this, "VisitCXXMemberCallExpr");
  const clang::CXXRecordDecl* rec = expr->getRecordDecl();
  const clang::CXXMethodDecl* method = expr->getMethodDecl();

  std::string recName = rec->getQualifiedNameAsString();
  dbg() << "[ExpressionEvaluator] Call method '" << method->getNameAsString() << "' from '" << recName << "'" << std::endl;

  Value object;
  std::vector<Value> args;
  Value result;
  if (EvalSubexpr(expr->getImplicitObjectArgument(), object) && CalculateCallArgs(expr->getArgs(), expr->getNumArgs(), args))
  {
    m_evalResult = value_ops::CallMember(m_interpreter, object, method, args, result);
  }

  vScope.Submit(std::move(result));
}
```

Code injection



Code injection - основные проблемы

- Описание инжектируемого кода должно компилироваться обычным компилятором
- Нужен способ описания разного рода деклараций (в том числе с генерируемым содержимым)
- Code injection и compile-time-вычисления могут быть перемешаны
- Нужен способ эмуляции доступа к “новым” (инжектированным) декларациям

Code injection

```
for (auto& t : meta::reflect_type<Types ...>())
{
   [[gsl::suppress("inject", "$target=dst", "public")]]
   [name="Visit"](const decltype(meta::project_type(t))& obj) -> void
   {
       meta::project("GetDerived")()->
               $_project_member("Visit")(std::forward<meta::project("T0")>(obj));
   };
}
```

Code injection

```
for (auto& t : meta::reflect_type<Types
{
  [[gsl::suppress("inject", "$target=dst", "public")]]
  [name="Visit"](const decltype(meta::project_type(t))& obj) -> void
  {
     meta::project("GetDerived")()->
          $_project_member("Visit")(std::forward<meta::project("T0")>(obj));
  };
}
```

Атрибут, отмечающий code injection

Code injection

```
for (auto& t : meta::reflect_type<Types ...>())
{
  [[gsl::suppress("inject", "$target=dst", "public")]]
  [name="Visit"](const decltype(meta::project_type(t))& obj) -> void
  {
    meta::project("GetDerived")()->
        $_project_member("Visit")(std::forward<meta::project("T0")>(obj));
  };
}
```

Инжектирование нового метода

Code injection

```
for (auto& t : meta::reflect_type<Types ...>())
{
  [[gsl::suppress("inject", "$target=dst", "public")]]
  [name="Visit"](const decltype(meta::project_type(t))& obj) -> void
  {
    meta::project("GetDerived")()->
          $_project_member("Visit")(std::forward<meta::project("T0")>(obj));
  };
}
```

Преобразование метаданных в соответствующий тип

KASPERSKY lab

Code injection

```
for (auto& t : meta::reflect_type<Types ...>())
{
  [[gsl::suppress("inject", "$target=dst", "public")]]
  [name="Visit"](const decltype(
  {
    meta::project("GetDerived")()->
         $_project_member("Visit")(std::forward<meta::project("T0")>(obj));
  };
}
```

Доступ к инжектированной сущности

Code injection

```
for (auto& t : meta::reflect_type<Types ...>())
{
   [[gsl::suppress("inject", "$target=dst", "public")]]
   [name="Visit"](const decltype(meta::project_type(t))& obj) -> void
   {
      meta::project("GetD
            $_project_member("Visit")(std::forward<meta::project("T0")>(obj));
   };
}
```

Доступ к методу инжектированной сущности.

Code injection

```
for (auto& t : meta::reflect_type<Types ...>())
{
   [[gsl::suppress("inject", "$target=dst", "public")]]
   [name="Visit"](const decltype(meta::project_type(t))& obj) -> void
   {
      meta::project("GetDerived")()->
              $_project_member("Visit")(std::forward<meta::project("T0")>(obj));
   };
}
```

Индектирование произвольного текста

Code injection

- clang::Rewriter для замены одних фрагментов кода другими
- Стек инжектирования

Code injection - атрибутирование

```
[[gsl::suppress("inject", "$target=dst", "public")]]
```

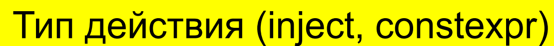
Code injection - атрибутирование

```
[[gsl::suppress("inject", "$target=dst", "public")]]
```

Специальный атрибут из набора clang

Code injection - атрибутирование

```
[[gsl::suppress("inject", "$target=dst", "public")]]
```



Code injection - атрибутирование

```
[[gsl::suppress("inject", "$target=dst", "public")]]
```

Code injection - стек

```
inline void BoostSerializable(meta::ClassInfo dst, const meta::ClassInfo& src)
{
  for (auto& v : src.variables())
     $_inject_v(dst, public) v;

  $_inject_v(dst, public) [&, name="serialize"](auto& ar, unsigned int ver) -> void
  {
     $_constexpr for (auto& v : src.variables())
        $_inject(_) ar & $_v(v.name());
  };
};
```

Code injection - стек

```
class TestStruct {
public:
   template <typename T0> void serialize(T0 &ar, const unsigned int ver) {
       ar & a;
       ar & b;
   }
   int a;
   std::string b;
};
```

Code injection - стек

```
inline void BoostSerializable(meta::ClassInfo dst, const meta::ClassInfo& src)
{
  for (auto& v : src.variables())
     $_inject_v(dst, public) v;

  $_inject_v(dst, public) [&, name="serialize"](auto& ar, unsigned int ver) ->
void
  {
     $_constexpr for (auto& v : src.variables())
        $_inject(_) ar & $_v(v.name());
  };
};
```

Инжектируется в dst в виде метода

Code injection - стек

```
inline void BoostSerializable(meta::ClassInfo dst, const meta::ClassInfo& src)
{
  for (auto& v : src.variables())
    $_inject_v(dst, public) v;

  $_inject_v(dst, public) [&, name="serialize"](auto& ar, unsigned int ver) -> void
  {
    $_constexpr for (auto& v : src.variables())
      $_inject(_) ar & $_v(v.name());
  };
};
```

Интерпретируется

Code injection - стек

```
inline void BoostSerializable(meta::ClassInfo dst, const meta::ClassInfo& src)
{
   for (auto& v : src.variables())
      $_inject_v(dst, public) v;

   $_inject_v(dst, public) [&, name="serialize"](auto& ar, unsigned int ver) -> void
   {
      $_constexpr for (auto& v : src.variables())
         $_inject(_) ar & $_v(v.name());
   };
};
```

Инжектируется в тело метода

Ещё пример

```
// Declare template metaclass for visitors across 'Types' list
template<typename R, typename ... Types>
inline void Visitor(meta::ClassInfo dst, const meta::ClassInfo& src)
{
  // Insert all methods form src to dst
  for (auto& f : src.functions())
     $_inject_v(dst, public) f;

  // Add extra 'Visit' methods to dst dependent on specific type from 'Types'
  for (auto& t : t_$(Types ...))
     $_inject_v(dst, public) { R Visit(const $_t(t)& obj); }
}
```

Ещё пример

```
$_class(SomeVisitor, Visitor<bool, A, B, C>)
{
public:
   void TestMethod1();
   std::string TestMethod2(int param) const;
};
```

Ещё пример

```
class SomeVisitor {
public:
    void TestMethod1();
    std::string TestMethod2(int param) const;
    bool Visit(const A &obj);
    bool Visit(const B &obj);
    bool Visit(const C &obj);

protected:
private:
};
```

Ещё пример

```
inline void Interface(meta::ClassInfo dst, const meta::ClassInfo& src) {
  using meta::compiler;
  compiler.require(src.variables().empty(), "Interface may not contain data members");

  $inject_v(dst, public) {
    enum {InterfaceId = 123456};
  }

  for (auto& f : src.functions()) {
    compiler.require(f.is_implicit() || (!f.is_copy_ctor() && !f.is_move_ctor()),
      "Interface can't contain copy or move constructor");

    if (!f.is_implicit()) {
      f.make_public();
      compiler.require(f.is_public(), "Inteface function must be public");
      f.make_pure_virtual();
      $_inject(dst) f;
    }
  }
}
```

Ещё пример

```
$_class(SomeVisitor, Visitor<bool, A, B, C>, Interface)
{
public:
   void TestMethod1();
   std::string TestMethod2(int param) const;
};
```

Ещё пример

```
class SomeVisitor {
public:
   virtual void TestMethod1() = 0;
   virtual std::string TestMethod2(int param) const = 0;
   virtual bool Visit(const A &obj) = 0;
   virtual bool Visit(const B &obj) = 0;
   virtual bool Visit(const C &obj) = 0;

protected:
private:
};
```

Ещё пример

```
class SomeVisitor {
public:
   void TestMethod1();
   std::string TestMethod2(int param) const;
   bool Visit(const A &obj);
   bool Visit(const B &obj);
   bool Visit(const C &obj);

protected:
private:
};
```

Полезные ссылки

- Спецификация метаклассов: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0707r3.pdf
- Ссылка на эти слайды: https://docs.google.com/presentation/d/1bWQXc1L51tMoK94AJFvpnQ7jVEjpinLR1Y_ELfzr4AY/edit?usp=sharing
- Репозиторий с кодом процессора метаклассов: https://github.com/flexferrum/autoprogrammer/
- Порт Jinja2 на C++: https://github.com/flexferrum/Jinja2Cpp

Спасибо!

telegram: @flexferrum

E-mail: flexferrum@gmail.com

Metaclasses processing tool - фронтенд

$_class(Name, MetaclassName...) - макрос, с помощью которого объявляется инстанс метакласса
$_inject_v(Target, Visibility) - макрос-префикс, который приводит к вставке следующего за ним блока в тело инстанса метакласса
$_constexpr() - макрос-префикс, помечающий следующий за ним оператор как вычисляемый на этапе компиляции
$_inject_xxx(Target, Name, Visibility), где XXX - это enum, struct, class или union - макрос-префикс, который приводит к вставке следующего за ним блока в тело инстанса метакласса в виде отдельной структуры, enum'а, класса или объединения
$_n(expr) - макрос, который заменяется значением выражения expr, вычисленного на этапе компиляции
$_t(expr) - макрос, который заменяется типом выражения expr, вычисленного на этапе компиляции
t_$(expr) - макрос, который заменяется метаинформацией о типе expr
$_str(expr) - макрос, который заменяется на expr

Metaclasses processing tool - фронтенд

meta::compiler - экземпляр класса meta::CompilerImpl, предоставляющий доступ к глобальным методам типа require, error, message и т. п.
meta::TypeInfo - класс, предоставляющий различную информацию о конкретном типе
meta::MemberInfo - класс, предоставляющий различную информацию о конкретном члене класса
meta::MethodInfo - класс, предоставляющий различную информацию о конкретном методе класса
meta::ClassInfo - класс, предоставляющий различную информацию о конкретном классе.

AST Reflector

Создаёт упрощённую модель AST с агрегированной информацией о:

- Классах/структурах (**reflection::ClassInfo**)
- Вложенных в класс декларациях (**reflection::ClassInfo::InnerDecl**)
- Методах и их параметрах (**reflection::MethodInfo**)
- Перечислений (**reflection::EnumInfo**)
- Типах (**reflection::TypeInfo**)
  - built-in/template/enum/record/alias/array/…
  - cv-qualifiers
  - pointing levels
  - name
  - …
- Пространствах имён (**reflection::NamespaceInfo**)

AST Reflector - пример

```
struct MethodInfo : public NamedDeclInfo
{
   SourceLocation declLocation;
   SourceLocation defLocation;
   std::vector<MethodParamInfo> params;
   std::vector<TemplateParamInfo> tplParams;
   std::string fullPrototype;
   TypeInfoPtr returnType;
   std::string returnTypeAsString;
   AccessType accessType = AccessType::Undefined;
   std::string body;

   bool isTemplate() const {return !tplParams.empty();}

   const clang::FunctionDecl* decl;


   ...
```

```
...
   bool isConst = false;
   bool isVirtual = false;
   bool isPure = false;
   bool isNoExcept = false;
   bool isRVRef = false;
   bool isCtor = false;
   bool isDtor = false;
   bool isOperator = false;
   bool isImplicit = false;
   bool isDeleted = false;
   bool isStatic = false;
   bool isExplicitCtor = false;
   bool isInlined = false;
   bool isClassScopeInlined = false;
   bool isDefined = false;
   AssignmentOperType assignmentOperType =
AssignmentOperType::None;
   ConstructorType constructorType = ConstructorType::None;
};
```

Интерпретатор - пример

```
case Stmt::IfStmtClass:
{
  const IfStmt *ifStmt = cast<IfStmt>(stmt);

  BlockScopeRAII scope(m_scopes);
  if (const Stmt *initStmt = ifStmt->getInit())
  {
     auto result = ExecuteStatement(initStmt, nullptr);

     if (result != ESR_Succeeded)
        return result;
  }
  bool cond = false;

...
```

```
...
  auto condVarStmt = ifStmt->getConditionVariable();
  if (condVarStmt && !ExecuteVarDecl(condVarStmt))
     return ESR_Failed;

  if (!ExecuteAsBooleanCondition(ifStmt->getCond(), cond))
     return ESR_Failed;

  if (const Stmt *subStmt = cond ? ifStmt->getThen() :
ifStmt->getElse())
  {
     auto result = ExecuteStatement(subStmt, nullptr);
     if (result != ESR_Succeeded)
        return result;
  }
  return ESR_Succeeded;
}
```

Вычислитель выражений - static reflection

```
static auto thunk = [](auto fnPtr, InterpreterImpl* interpreter, Value& obj, const std::vector<Value>& args, Value& result) {
  return ApplyCallMemberVisitor(fnPtr, obj, interpreter, args, result);
};
using ThunkType = decltype(thunk);
using ThunkInvoker = std::function<bool (const ThunkType&, InterpreterImpl*, Value&, const std::vector<Value>&, Value&)>;

static auto thunkMaker = [](auto fnPtr) -> ThunkInvoker {
  return [fnPtr](const ThunkType& thunk, InterpreterImpl* interpreter, Value& obj, const std::vector<Value>& args, Value& result) ->
bool {
     return thunk(fnPtr, interpreter, obj, args, result);
  };
};
```

Вычислитель выражений - static reflection

```
template<typename Fn>
struct CallMemberVisitor : boost::static_visitor<bool> {

  template<typename U, typename T, typename ... Args, size_t ... Idxs>
  bool Invoke(bool (*fn)(InterpreterImpl*, T, Value&, Args...), U&& val, const std::index_sequence<Idxs...>&) const {
     return fn(interpreter, std::forward<U>(val), result, ConvertValue<std::decay_t<Args>>(args[Idxs])...);
  }

  template<typename U, typename T, typename ... Args>
  auto Call(bool (*fn)(InterpreterImpl*, T, Value&, Args...), U&& val) const -> decltype(fn(interpreter, std::forward<U>(val), result,
std::declval<Args>()...)) {
     return Invoke(fn, std::forward<U>(val), std::make_index_sequence<sizeof ... (Args)>());
  }

  template<typename U>
  auto operator()(U&& val) const -> decltype(Call(fn, std::forward<U>(val))) {
     return Call(fn, std::forward<U>(val));
  }
};
```

Code injection - обработка выражений

```
template<typename T>
struct Projector : public ProjectorBase
{
   using type = T;

   template<typename ... Args>
   Projector<T> operator()(Args&& ... args);

   template<typename R>
   operator R();

   template<typename U>
   Projector<T> $_project_member(U&&);

   Projector<T>* operator ->();
};
```