C++ Russia

MyRTTI

(библиотека для динамической идентификации типа данных)

RTTI решение для C++ с полноценным доступом к иерархии классов

Зачем нам вообще нужен RTTI?

Пример

```
class Shape {/*...*/};
class Circle : public Shape {/*...*/};
class Square : public Shape {/*...*/};

void print(const vector<Shape*>& objects) {
   if (const auto* o: objects) {
      print(o);
   }
}

void print(const Shape* obj) {
   if (const auto* circle = cast<Circle>(obj)) {
      cout << “Круг” << endl;
      circle->printCenter();
      circle->printRadius();
   }
   if (const auto* square = cast<Square>(obj)) {
      cout << “Квадрат” << endl;
      square->printLeftTop();
      square->printWidth();
   }
}
```

Зачем нам вообще нужен RTTI?

Немного о наших целях
Мы не стремимся побить все рекорды производительности.

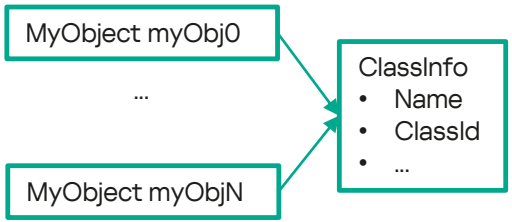
Наша цель – сделать удобный интерфейс с адекватной скоростью работы и потреблением RAM.

Пример

```
void print(const vector<Shape*>& objects) {
   if (const auto* o: objects) {
      print(o);
   }
}

void print(const Shape* obj) {
   if (const auto* circle = cast<Circle>(obj)) {
      cout << “Круг” << endl;
      circle->printCenter();
      circle->printRadius();
   }
   if (const auto* square = cast<Square>(obj)) {
      cout << “Квадрат” << endl;
      square->printLeftTop();
      square->printWidth();
   }
}
```

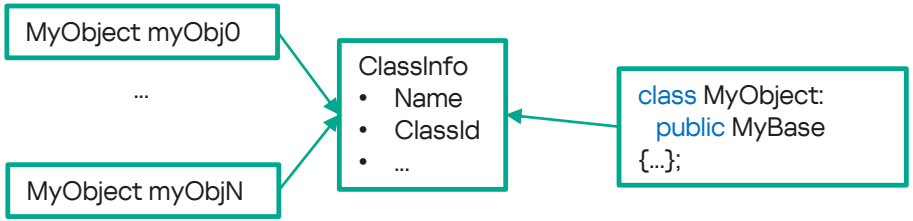
Что нам для этого нужно?

Что нам для этого нужно?



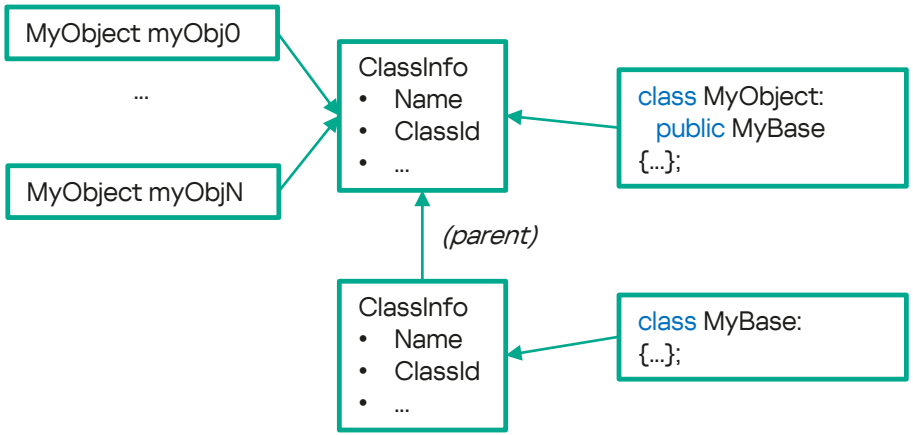
1. В каждый объект нужно добавить (runtime) информацию о его типе...
   ... в идеале о всех его родительских типах.

Что нам для этого нужно?



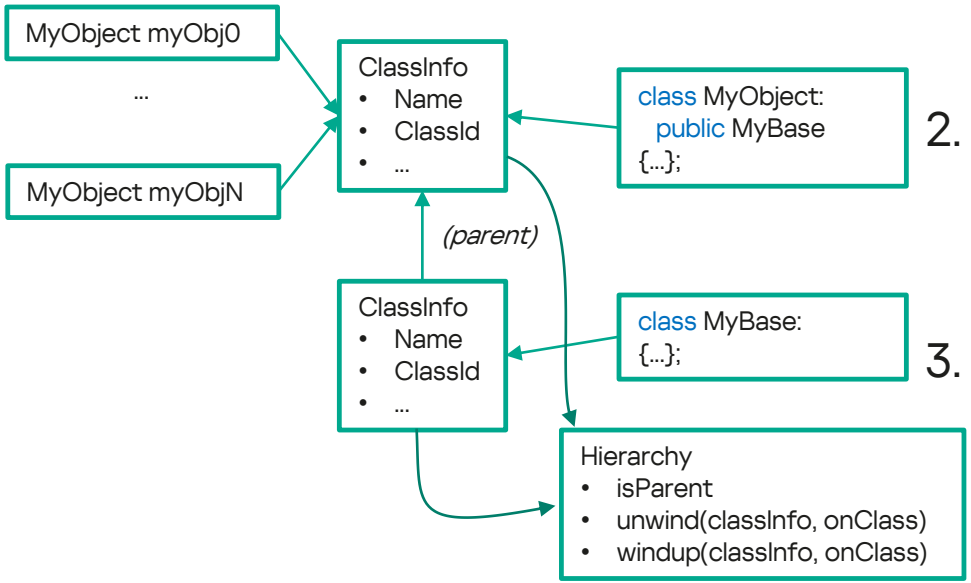
1. В каждый объект нужно добавить (runtime) информацию о его типе...
   ... в идеале о всех его родительских типах.
2. А еще надо связать эту информацию с классами (static members, constexpr)

Что нам для этого нужно?



1. В каждый объект нужно добавить (runtime) информацию о его типе...
   ... в идеале о всех его родительских типах.
2. А еще надо связать эту информацию с классами (static members, constexrpr)
3. Очень хочется иметь доступ к графу иерархии...

Что нам для этого нужно?



1. В каждый объект нужно добавить (runtime) информацию о его типе...
... в идеале о всех его родительских типах.
2. А еще надо связать эту информацию с классами (static members, constexpr)
3. Очень хочется иметь доступ к графу иерархии... Полноценный.

А что нам мешает?

Стандартная поддержка

1. Она применима глобально к translation unit.
2. Стандартный visit.. Это боль.
3. Узнать список parents невозможно.

Следствие: очень ограниченная область применения.

А что нам мешает?

«Своя» поддержка

1. Усложняется дизайн пользовательских классов.
2. У каждого решения свои достоинства и недостатки.

Стандартная поддержка

1. Она применима глобально к translation unit.
2. Стандартный visit.. Это боль.
3. Узнать список parents невозможно.

Следствие: очень ограниченная область применения.

Небольшой обзор

- LLVM, простой classof
- LLVM, для открытых иерархий (через RTTIExtends)
- Windows MFC
- Unreal Engine

LLVM, classof

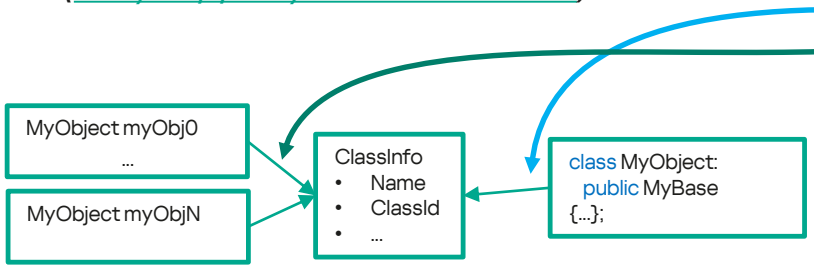
Достоинства
- Быстрый кастинг
- Очень понятный и простой дизайн всего решения

Недостатки
- При создании нового класса нужно:
- Модифицировать базовый класс (обновить enum)
- Особый конструктор в новом классе (нужно указать – кто мы)
- Нет поддержки иерархии, инфа хранится только о top-most классе.
- Visitor требует доп. кодогенерации (tablegen).
- Применим только если известен конечный набор классов.

```
class Shape {
public:
   /// Discriminator for LLVM-style RTTI (dyn_cast<> et al.)
   enum ShapeKind {
      SK_Square,
      SK_Circle
   };
private:
   const ShapeKind Kind;
public:
   ShapeKind getKind() const { return Kind; }
   Shape(ShapeKind K) : Kind(K) {}

   // class body
};

class Square : public Shape {
   double SideLength;
public:
   Square(double S) : Shape(SK_Square), SideLength(S) {}

   // class body
};

void usage() {
   Shape *S = ...;

   // "isa" picks S->Kind and compares it with SK_Circle
   if (isa<Circle>(S)) {/* do something ... */}
}
```

LLVM, RTTIExtends

(*llvm/Support/ExtensibleRTTI.h*)



Достоинства

- Относительно простой дизайн классов:
  - Наследуемся от `RTTIExtends<NewClass>`
  - Объявляем статическую `NewClass::ID`
- Не надо обновлять родительский класс

Недостатки

- Нет поддержки иерархии классов.
- Инициализация `NewClass::ID` в .cpp

```
// .h
template <typename ThisT, typename ParentT>
class RTTIExtends : public ParentT {
public:
   // Inherit constructors from ParentT.
   using ParentT::ParentT;
   static const void *classID() { return &ThisT::ID; }
   const void *dynamicClassID() const override
   { return &ThisT::ID; }

   bool isA(const void *const ClassID) const override {
      return ClassID == classID() || ParentT::isA(ClassID);
   }
};

class Shape : public RTTIExtends<Shape, RTTIRoot> {
public:
   static char ID;
   // class body
};

class Square : public RTTIExtends<Square, Shape> {
public:
   static char ID;
   // class body
};

// .cpp
char Shape::ID = 0;
char Square::ID = 0;
```

Windows MFC

Достоинства
- Хранит информацию об иерархии
- Не надо обновлять родительский класс

Недостатки
- При создании нового класса надо обновлять как .h так и .cpp файл
- Нет поддержки множественного наследования
- Косвенно, через *CRuntimeClass::m_pBaseClass* можно получить доступ к иерархии (см. CRuntimeClass)

```
class CPerson : public CObject
{
   // Макрос объявляет
   // virtual CRuntimeClass* GetRuntimeClass();
   // static CRuntimeClass* GetThisClass();
   // static CRuntimeClass* GetBaseClass();
   DECLARE_DYNAMIC(CPerson)

   // ...
};

// .cpp
// А этот макрос прописывает реализацию всего того,
// что объявил DECLARE_DYNAMIC.
IMPLEMENT_DYNAMIC(CPerson, CObject)
// ...
```

UnrealEngine, UObject

Достоинства

- Хранит информацию об иерархии, похоже на `OSMetaClass`, но еще появляются всякие reflection штуки (`UProperty`, `UEnum` итд)
- Все доп. объявления надо делать только в .h файле
- Не надо обновлять родительский класс

Недостатки

- Есть сгенерированный код
- Нет поддержки множественного наследования
- Если переиспользовать, то, наверное, будут проблемы с лицензией

```
// .h
#pragma once

#include 'Object.h'
#include 'MyObject.generated.h'
UCLASS()
class MYPROJECT_API UMyObject : public UObject
{
   GENERATED_BODY()
public:
   // ...
};
```

Наше решение: MyRTTI

- Дополнительные объявления только в заголовке класса. Но пока все таки в двух (смежных) местах.
- Если использовать «тяжелые» макросы, то можно сделать объявление в одной строке.
- Не надо менять родительский класс.
- Не надо менять конструкторы.
- Есть ограниченная поддержка множественного наследования.

Как это выглядит?

```cpp
#include <myrtti.h>

struct Shape : myrtti::RTTI<Shape>
{
    DEFINE_RTTI(Shape, myrtti::Object);

    // class body
    // ...
};

struct Square : Shape, myrtti::RTTI<Square>
{
    DEFINE_RTTI(Square, Shape);

    // class body
    // ...
};
```

```cpp
#include <iostream>
#include <memory>
#include <vector>

#include "figures.h"

using namespace std;

int main() {
    vector<shared_ptr<myrtti::Object>> objs = {
        make_shared<Shape>(),
        make_shared<Square>(),
    };

    for (auto &o : objs) {
        cout << "o->rtti->name = " << o->rtti->name << "\n";

        if (myrtti::isa<Square>(o)) {
            cout << "Discovered Square\n";
        }
    }
    return 0;
}
```

Как это выглядит?

(через with_rtti_xxx)

```
#include <myrtti.h>

with_rtti_root(struct, Shape)
    // Structure body here
    // ...
    private:
    // Private members here
    // ...
with_rtti_end();

with_rtti(struct, Square, Shape)
    // body
    // ...
    private:
    // Private members here
    // ...
with_rtti_end();
```

```
#include <iostream>
#include <memory>
#include <vector>

#include "figures.h"

using namespace std;

int main() {
    vector<shared_ptr<myrtti::Object>> objs = {
        make_shared<Shape>(),
        make_shared<Square>(),
    };

    for (auto &o : objs) {
        cout << "o->rtti->name = " << o->rtti->name << "\n";

        if (myrtti::isa<Square>(o)) {
            cout << "Discovered Square\n";
        }
    }
    return 0;
}
```

Как устроено?

(`RTTI<T>` и пользовательские классы)

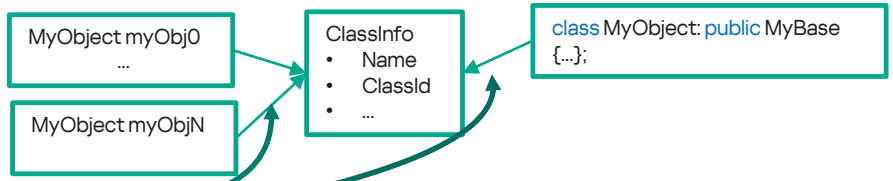
Как устроено?

(RTTI<T> и пользовательские классы)

```
namespace myrtti {

// Пользовательские классы NewClass
// наследуются от RTTI<NewCLass>
//
// Именно этот механизм позволяет автоматически
// встраивать runtime информацию о типе
// через вызов статического метода-синглтона
// NewClass::info()
template <class Class>
struct RTTI : virtual Object {
    RTTI() {
        auto *s = static_cast<Class*>(this);
        this->rtti = Class::info();
        this->crossPtrs[Class::class_id()] = s;
    }
};
}
```

Как устроено?

(RTTI<T> и пользовательские классы)

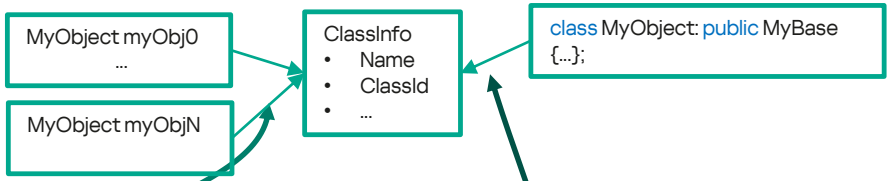



```
namespace myrtti {

// Пользовательские классы NewClass
// наследуются от RTTI<NewCLass>
//
// Именно этот механизм позволяет автоматически
// встраивать runtime информацию о типе
// через вызов статического метода-синглтона
// NewClass::info()
template <class Class>
struct RTTI : virtual Object {
    RTTI() {
        auto *s = static_cast<Class*>(this);
        this->rtti = Class::info();
        this->crossPtrs[Class::class_id()] = s;
    }
};
}
```

```
namespace myrtti {

// Еще мы должны в пользовательских классах объявить
// * class_id (это constexpr CRC64 от его имени)
// * сам метод info()
#define DEFINE_RTTI(cn, ...)                                          \
    static MYRTTI_INLINE constexpr myrtti::class_id_t class_id() { \
        constexpr myrtti::class_id_t myId{MYRTTI_UNIQUE_NAME(cn)}; \
        return myId;                                                  \
    }                                                                 \
    static const ::myrtti::ClassInfo* info()     {                  \
        static std::unique_ptr<myrtti::ClassInfo>                     \
        p = myrtti::ClassInfo::create<cn, __VA_ARGS__>(#cn);          \
        return p.get();                                               \
    }
}
```

Как устроено?

(RTTI<T> и пользовательские классы)




```
// Пользовательский класс после всех подстановок выглядит вот так:
struct Square : Shape, RTTI<Square>
{
    // Раскрытие макроса DEFINE_RTTI
    static MYRTTI_INLINE constexpr myrtti::class_id_t class_id() {
        constexpr myrtti::class_id_t myId{"Square", "file.cpp", 32};
        return myId;
    }
    static const ::myrtti::ClassInfo *info() {
        static std::unique_ptr<myrtti::ClassInfo>
        p = myrtti::ClassInfo::create<Square, Shape>("Square");
        return p.get();
    }
    // Далее идет «обычное» тело структуры или класса.
};
```

Как устроено?

(основные типы)

```
// Структура ClassInfo – точка входа RTTI
struct ClassInfo {
   const char* name;
   // ...

   template<class ThisClass, class ...Parents>
   static inline std::unique_ptr<ClassInfo> create(const char* name) {
       // Принудительно вызываем info() у всех родителей,
       // тем самым, создавая их ClassInfo если таковые еще
       // не созданы.
       ( [&] { Parents::info(); } (), ... );

       // Создаем наш собственный ClassInfo.
       return std::make_unique<ClassInfo>(
           name, ThisClass::class_id(), mk_class_ids<Parents...>()
       );
   }

   // Основной конструктор
   template<typename ArrayT>
   ClassInfo(
      const char* name,
      class_id_t classId,
      const ArrayT& parents
   )
   : name(name), id(classId) {
      Hierarchy::instance()->add(this, parents);
   }
   // ...
};
```

Как устроено?

(основные типы)


```cpp
namespace myrtti {

// Объект от которого мы явно (или неявно) должны унаследоваться
// если хотим иметь runtime type информацию.
struct Object {
    virtual ~Object() = default;
    const ClassInfo* rtti = nullptr;

    static MYRTTI_INLINE constexpr class_id_t class_id() { /*...*/ }
    static const ClassInfo* info() { /*...*/ }

protected:
    // Здесь мы будем хранить указатели на «себя»
    // необходимые для cross-cast.
    // Вопрос: Почему map?!
    // Ответ: Мы пробовали unordered_map, и он подойдет если
    // у вас очень большая иерархия, но в случае коллизий
    // он начиниает уступать map и dynamic_cast.
    std::map<class_id_t, void*> crossPtrs{{class_id(), this}};

    // RTTI<T> - наш друг, у него должен быть доступ к crossPtrs.
    template<class T> friend class RTTI;

public:
    // Вариации cast (на самом деле их много, и шаблон там сложнее)
    template<class T>
    T* cast() {
        auto found = this->crossPtrs.find(T::class_id());
        if (found != end(this->crossPtrs)) {
            return static_cast<T*>(found->second);
        }
        return nullptr;
    }
};
```

Масштабируемость

(и вообще с чем можно поиграться)

Иерархия

- Можно и без нее, оставить только class_id, и тогда по функционалу это будет аналог LLVM classof. Сравнение “isa” будет таким же быстрым: это тоже сравнение с константой.
- Дизайн класса в этом случае можно сильно упростить.
- И нам по-прежнему не надо править родительский класс.

```
// Выглядеть это будет примерно вот так:
struct Square : Shape, RTTI<Square>
{
    static constexpr class_id_t class_id =
        class_id_t("Square");
};
```

Масштабируемость

(и вообще с чем можно поиграться)

cast

- ...Можно убрать crossPtrs

- ...или (если single inheritance) crossPtrs заменить на

```
vector<class_id_t> generationToClass;
```

и еще добавить вот это:

```
static constexpr int generation =
Parent::generation+1;
```

тогда

```
isa -> return
generationToClass[T::generation] ==
T::class_id;
```

```
namespace myrtti {

struct Object {
    virtual ~Object() = default;
    static constexpr class_id_t class_id{"myrtti::Object"};
    class_id_t myId = class_id;

protected:

    // RTTI<T> - по-прежнему - наш друг, у него должен
    // быть доступ к myId,
    // чтоб каждый новый потомок вписать туда свои данные.
    template<class T> friend class RTTI;

public:
    // Вариации cast
    template<class T>
    T* cast() {
        // Мы экономим на вызове вот этого метода
        // auto found = this->crossPtrs.find(T::class_id);
        if (T::class_id == myId) {
            return static_cast<T*>(this);
        }
        return nullptr;
    }
};
```

Что можно улучшить

- Оптимизация когда не используется множественное наследование

- Оптимизация случаев, когда можно сделать *static_cast:*
  - Для всех классов делаем не-виртуальную перегрузку *cast* вот с такой надстройкой:

```
if constexpr (ThisT::static_parents.count(TargetT::class_id()))
    return static_cast<TargetT*>(this);
```

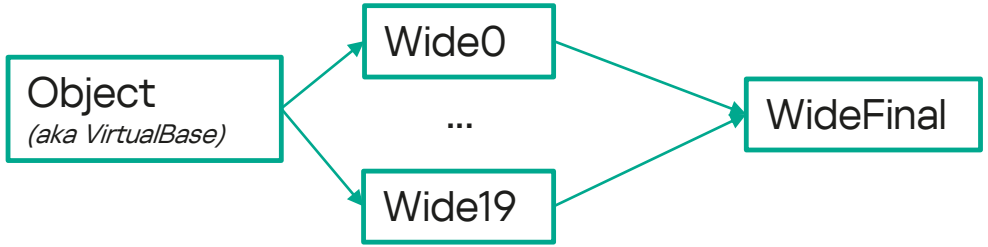
- Можно написать плагины для clang и gcc (добавить что-то вроде *[[myrtti::add_runtime]]*)

Производительность

Что проверяли?

«Глубокая» иерархия (сравнили с dynamic_cast, MFC, UE4)

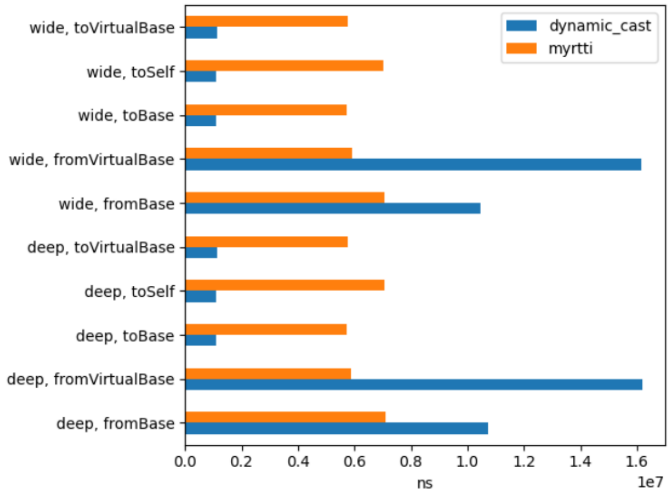
«Широкая» иерархия (сравнили с dynamic_cast)



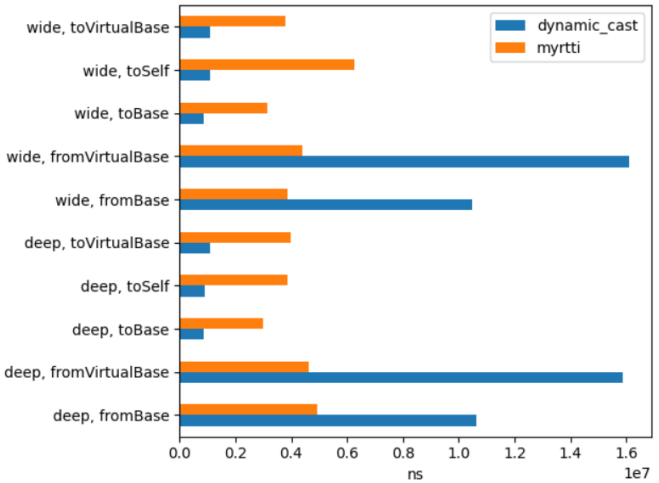
Производительность

myrtti vs dynamic_cast

Clang 10.0.0 -O3



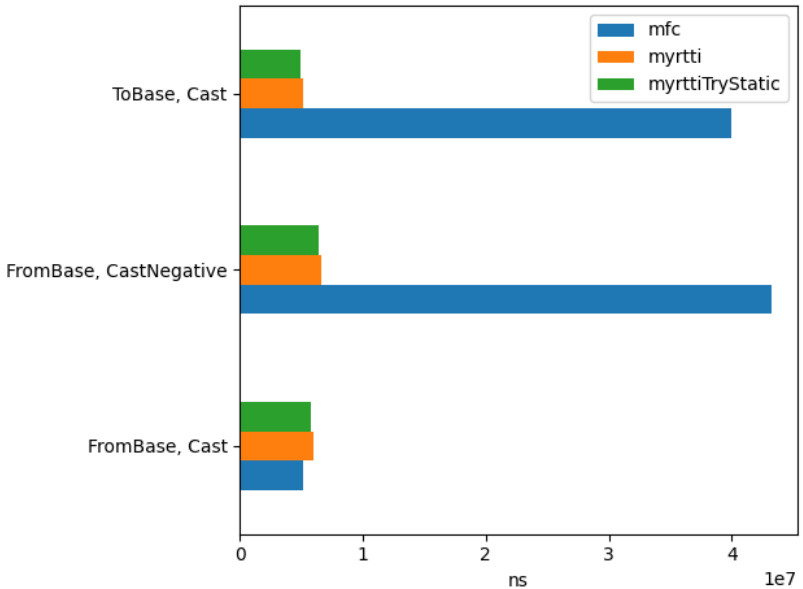
GCC 9.4.0 -O3



(чем длиннее полоска – тем медленнее решение)

Производительность

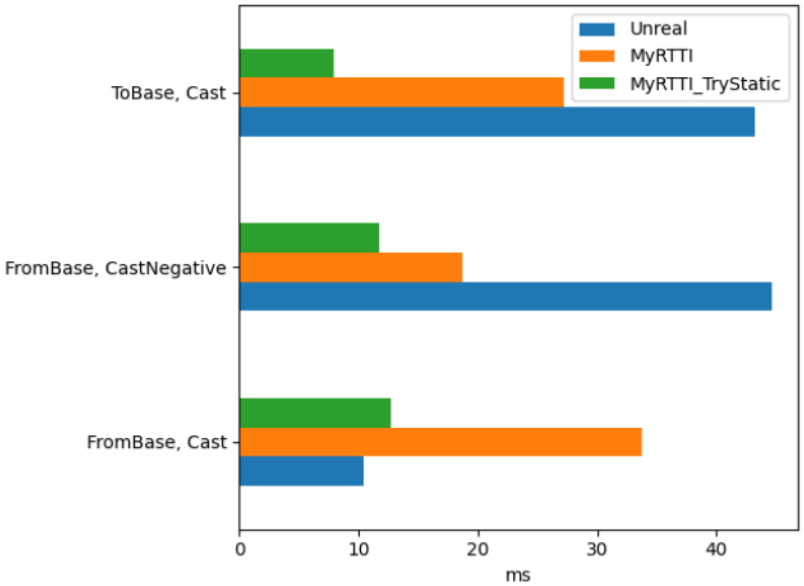
myrtti vs MFC



(чем длиннее полоска – тем медленнее решение)

Производительность

myrtti vs Unreal Engine



(чем длиннее полоска – тем медленнее решение)

Производительность

Некоторые выводы

Про позитивные моменты
...отнесемся к ним скептически.

Наше решение *вроде как*
быстрее при конвертации
«*база -> потомок*», но...

- Тесты были «синтетические»
и их объективно – мало. Нужно
больше различных архитектур
и компиляторов.

Производительность

Некоторые выводы

Про позитивные моменты
...отнесемся к ним скептически.

Наше решение *вроде как* быстрее при конвертации «*база -> потомок*», но...

- Тесты были «синтетические» и их объективно – мало. Нужно больше различных архитектур и компиляторов.

Про места где наше решение проиграло dynamic_cast.
...воспримем их всерьез ☺.

Стандартом (ссылка на черновик, раздел 5.2.7, параграф 5) описаны случаи когда *dynamic_cast* заменяется на *static_cast*.

Они, конечно же, учтены компиляторами, но не учтены нами.

Производительность

Некоторые выводы

Про места где наше решение проиграло MFC и Unreal Engine.

Оба сторонних решения оптимизированы под случай, когда нет множественного наследования.

Теоретически – мы тоже можем определять такие случаи и использовать для них оптимизированные алгоритмы.

myrtti::Visitor

Пример реализации type-matching visitor

Примеры других решений:

- LLVM RecursiveASTVisitor
  (вот тут вставляется сгенерированный код)
- std::visit

Как это выглядит?

```cpp
#include <iostream>

#include "myrtti.h"
#include "myrtti/visitor.h"
#include "user_defined_exceptions.h"

using namespace std;
using namespace myrtti;

int main() {

    std::cout << "Checking classic visitor...\n";

    Visitor visitor(
        [](const Exception& e) {
            cout << "TEST EXCEPTION: Exception, msg: " << e.message << ".\n";
            return true;
        },
        [](const ExceptionErrorOne& e) {
            cout << "TEST EXCEPTION: ExceptionOne.\n";
            return true;
        },
        [](const ExceptionErrorTwo& e) {
            cout << "TEST EXCEPTION: ExceptionTwo.\n";
            return true;
        }
    );

    cout << "Running classic visitor:\n";

    visitor.visit(ExceptionErrorOne());
    visitor.visit(ExceptionErrorTwo());
    visitor.visit(ExceptionErrorThree());

    return 0;
}
```

Вывод

```
TEST EXCEPTION: ExceptionOne.
TEST EXCEPTION: ExceptionTwo.
TEST EXCEPTION: Exception, msg: Exception Three Error.
```

Как работает?

Конструктор

```
struct Visitor {
   template<class ...Lambda>
   explicit Visitor(Lambda&& ...L) {
      // Трюк для распаковки типов параметров лямбды
      init(std::function(L)...);
   }

   template<class ...Cls>
   void init(

      // Тут мы уже знаем все связки «тип + обработчик».
      // От типа нам в данном случае нужен только тип::class_id.
      std::function<bool(Cls&)>&& ...visitors
   ) {
      (
         [&] {
            visitorsMap.emplace(Cls::class_id(),
               [=] (Object& b) {
                     Cls& bb = b.template cast<Cls>();
                     return visitors(bb);
               }
            );
         } (), ...
      );
   }
   // ...
};
```

Как работает?

Конструктор

```
struct Visitor {
   template<class ...Lambda>
   explicit Visitor(Lambda&& ...L) {
      // Трюк для распаковки типов параметров лямбды
      init(std::function(L)...);
   }

   template<class ...Cls>
   void init(

      // Тут мы уже знаем все связки «тип + обработчик».
      // От типа нам в данном случае нужен только тип::class_id.
      std::function<bool(Cls&)>&& ...visitors
   ) {
      (
         [&] {
            visitorsMap.emplace(Cls::class_id(),
               [=] (Object& b) {
                     Cls& bb = b.template cast<Cls>();
                     return visitors(bb);
               }
            );
         } (), ...
      );
   }
   bool visit(Object& b) {/* … */}

   // ...
};
```

Visitor::visit

```
bool visit(Object& b) {
   // Пользуемся тем, что у нас есть полноценный доступ
   // к графу иерархии:
   // делаем обход в глубину, пока не найдем первый доступный
   // обработчик.
   bool neverVisited = Hierarchy::instance()->unwind(
      b.rtti->getId(),
      [&] (const ClassInfo* cls) {
            auto found = visitorsMap.find(cls->getId());
            if (found != end(visitorsMap)) {
               // С учетом специфики Hierarchy::unwind
               // нам нужно вернуть false, чтобы закончить
               // обход.
               bool visitHandled = found->second(b);
               return !visitHandled;
            }
            return true;
      }
   );
   // Если unwind вернул нам false, значит обработчик
   // был найден, и это - хорошо, поэтому мы возвращаем true.
   // И если наоборот, то возвращаем false.
   return !neverVisited;
}
```

Конец. Всем спасибо!

- Пока это скорее PoC, но нам, конечно, очень интересно будет работать над ним дальше.
- Весь исходный код можно посмотреть на github: myrtti:
  - Базовый пример
  - Visitor
- Обучающая C++ игра «Миссия Мидори» от Лаборатории Касперского (это реклама ☺)