CppCon 2017
Trip Report

Jean-Michel Carter

Faster Delivery of Large C/C++ Projects with Conan … (Diego Rodriguez-Losada)

- Conan : Gestion des Librairies et de leurs dépendances (en utilisant SemVer (Semantic Versioning i.e. Major.Minor.Incremental) à l'intérieur d'un projet.
- Upload les binaires des librairies sur un Serveur. (Par architecture/version)
- Conan contient un Serveur (Open Source - with MIT license)
- Pour plus de robustesse (Enterprise solution), Diego (auteur de Conan) suggère d'utiliser Artifactory (JFrog), pour gérer les binaires sur le serveur. Sinon, le client peut aussi simplement recompiler avec les sources, si disponibles.
- Possibilité de rendre le tout automatique avec une intégration continue, par example, avec Jenkins (Tous les examples de la démo sont fait avec Artifactory et Jenkins).

Mocking Framework considered harmful (Peter Sommerlad)

- Problème avec les Mocking Framework : Doit “extraire” une interface polymorphique manuellement, et apprendre un paquet de “Magical Macros” pour les Mocks (au lieu de code C++ régulier).

- Mockator (intégré à Cevelop : IDE basé sur Eclipse CDT développé par le présentateur, Peter Sommerlad), permet d’extraire une interface automatiquement, et de générer du code pour les mocks de façons automatique.

C++ Mocking Frameworks Problems

- Design tend to follow JMock/EasyMock - keep Java Design- lack C++ strengths
- No useful reflection in C++: Macros for generating names, defining function stubs
- Subclassing, virtual member functions, sometimes fiddling via undefined behavior or low-level machine-specific ABI, e.g., replacing vtable entries (Hippomocks)
- DSL to specify (expected) behavior, not plain code, MACRO magic
  - with implicit matching of actual behavior vs expected in destructors
  - fragile with refactoring, hard to reuse across tests, bloated tests

```
EXPECT_CALL(turtle, GetY())
    .WillOnce(Return(100))
    .WillOnce(Return(200))
    .WillRepeatedly(Return(300));
```

Simpler Mocking with Mockator

- NO #define MACROS for users (almost)
- Introduce Seam refactorings

```
MockWarehouse warehouse { };
OrderT<MockWarehouse> order(TALISKER,50);
order.fill(warehouse);
ASSERT( order.isFilled());
calls expectedMockWarehouse{
   call("MockWarehouse()"),
   call("hasInventory(const std::string&, int) const", TALISKER,50),
   call("remove(const std::string&, int) const", TALISKER,50)
};
ASSERT_EQUAL(expectedMockWarehouse, allCalls[1]);
```

- Generate **regular C++** code for Seam by IDE (Cevelop)
- Generate **regular C++** code for tracing calls by IDE (Cevelop) on demand
- Use vector<call> for tracing calls (call ~ std::string)
- Use C++ and std::regex for matching, if needed

see videos at http://mockator.com

Summary

- If you need to test legacy code - introduce seams and stub DOCs
  - Take all the power C++ and IDEs provide to simplify that as needed
- Only if you have to test the use of a non-changeable stateful API - use mocks
- Be aware of the dangers of mocking (frameworks) for new code's design
- Do not mock unwritten code with a mocking framework (premature design)
- Remember: KISS applies to test automation as well

10 Core Guidelines you need to start using now (Kate Gregory) (https://github.com/isocpp/CppCoreGuidelines)

1. C.45 : Don't define a default constructor that only initializes data members; use in-class member initializers instead.
   C.48 : Prefer in-class initializers to member initializers in constructors for constant initializers
2. F.51: Where there is a choice, prefer default arguments over overloading.
3. C.47: Define and initialize member variables in the order of member declaration.
4. I.23: Keep the number of function arguments low
5. ES.50: Don't cast away const.
6. I.11: Never transfer ownership by a raw pointer (T*) (gsl::owner<T*>)
7. F.21: To return multiple "out" values, prefer returning a tuple or struct.
8. Enum.3: Prefer class enums over "plain" enums
9. I.12: Declare a pointer that must not be null as not_null. (gsl::not_null<T*>)
10. ES.46: Avoid lossy (narrowing, truncating) arithmetic conversions (gsl::narrow_cast<>)

RunTime Polymorphism (Louis Dionne)

- Différentes options pour le polymorphisme en Runtime.
- https://github.com/ldionne/dyno

I JUST WANTED THIS!

```
interface Vehicle { void accelerate(); };

namespace lib {
  struct Motorcycle { void accelerate(); };
}
struct Car   { void accelerate(); };
struct Truck { void accelerate(); };

int main() {
  std::vector<Vehicle> vehicles;
  vehicles.push_back(Car{...});
  vehicles.push_back(Truck{...});
  vehicles.push_back(lib::Motorcycle{...});

  for (auto& vehicle : vehicles) {
    vehicle.accelerate();
  }
}
```

RunTime Polymorphism (Louis Dionne)

- Problèmes avec l'Héritage : 1. Intrusif. 2. Incompatible avec le "value semantics". 3. Couplé avec le storage dynamique. 4. LENT!

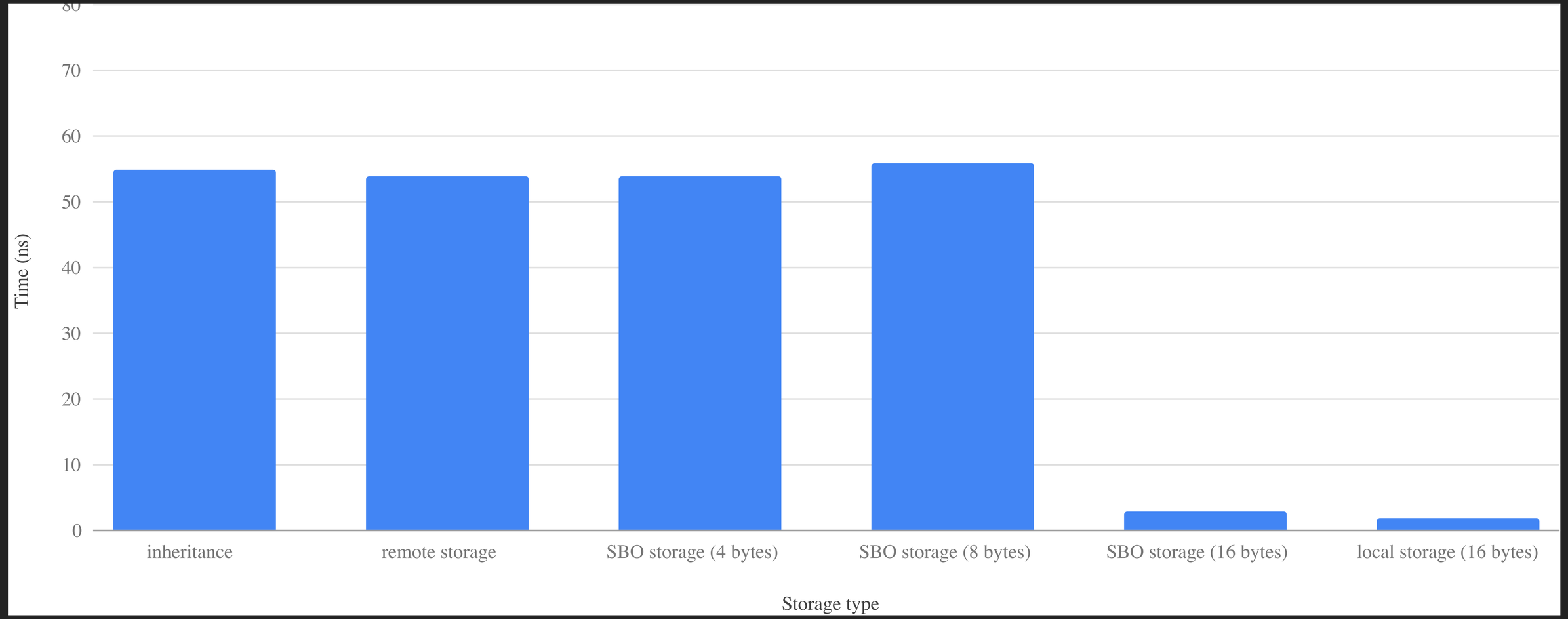
- Compare Héritage pour le polymorphisme avec une façon de gérer la politique de storage d'un objet polymorphique et la politique des dispatch d'appels de fonctions. (Storage Policy vs VTable Policy).

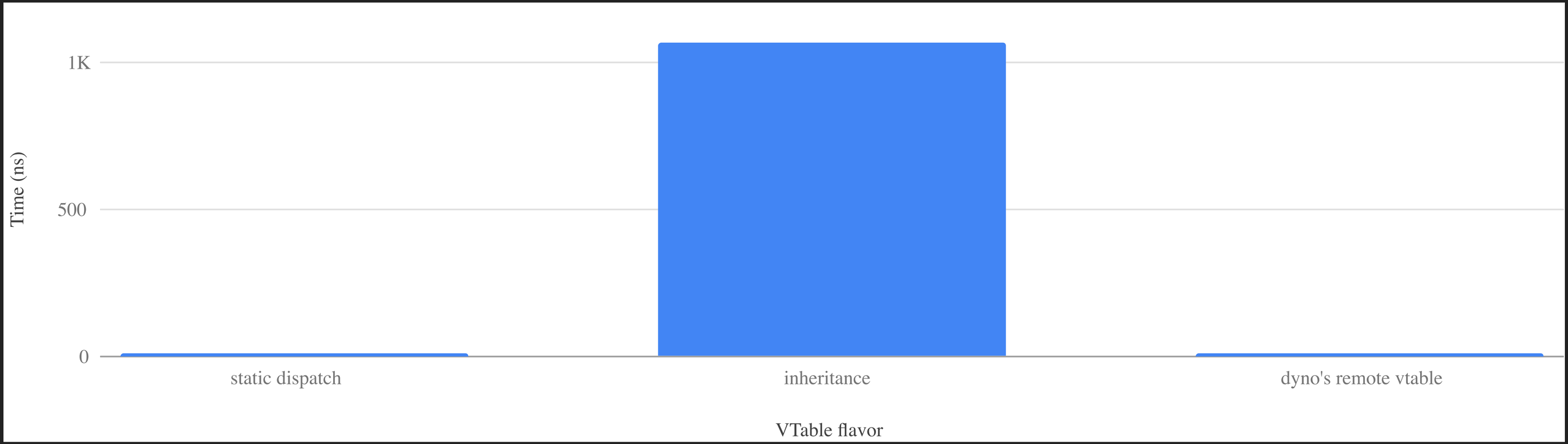
- 4 Options pour Storage : Héritage, Remote Storage, SBO, Always-Local Storage. Dyno supporte chacune de ces options.

- 4 Options pou VTable : Héritage, Inlining VTable, Partial Inlining (for static dispatch), Remote VTable.

SOME BENCHMARKS

Creating many 16 bytes objects




1K
500
0
Time (ns)
static dispatch
inheritance
dyno's remote vtable
VTable flavor

HAVE YOU HEARD OF THE FOLLOWING?

- `std::function`
- `inplace_function`
- `function_view`

HERE'S ALL OF THEM:

```
template <typename Signature>
using function = basic_function<Signature,
                                dyno::sbo_storage<16>>;

template <typename Signature, std::size_t Size = 32>
using inplace_function = basic_function<Signature,
                                        dyno::local_storage<Size>>;

template <typename Signature>
using function_view = basic_function<Signature,
                                     dyno::non_owning_storage>;

template <typename Signature>
using shared_function = basic_function<Signature,
                                       dyno::shared_remote_storage>;
```

Notables mais non présenté

- Microcontrollers in Micro-increments : A Test-driven C++ Workflow for Embedded Systems (Mike Ritchie)
- Building for the Best of Us: Design and Development with Kids in Mind (Sara Chipps)
- Traveling the Solar System with C++: Programming Rocket Science (Juan Arrieta)
- PostModern C++ (Tony Van Eerd)

Références

- CppCon Channel : CppCon 2017 Playlist on Youtube. https://www.youtube.com/channel/UCMlGfpWw-RUdWX_JbLCukXg

- GitHub CppCon (Presentation Slides) https://github.com/CppCon/CppCon2017