IoC++

A Inversion of Control library for C++

Sebastian Redl
C++Now 2016

About Me

- Work for TEOCO, Vienna office
- C++ programmer for 18 years
- Maintainer of Boost.PropertyTree

Background

- Cover an area with new mobile network transmitters
- Try to find minimal-cost solution while reaching certain quality goals
- Extremely large search space
- Use a combination of different heuristics

Background

- Need to experiment with different combinations of heuristics
- Want to build the algorithm from building blocks
- Define via configuration file
- Solution: Use dependency injection library

Background

- When we started, there were no good IoC libraries
  - No C++11 support
  - Some crash when used with multiple inheritance hierarchies
- Colleague of mine started IoC++
- This is the fourth iteration

Features

- Complete type erasure (DI container is not a template)
- Separate compilation of different parts of container configuration
- User-extensible lifetimes
- User-extensible argument value conversion
- Context-dependent mapping from interface to implementation

Basic Usage

- Create a Container
- Feed it reflection information about your classes, the interfaces they implement, and the constructors they have
- Define named objects based on the types; associate their parameters with other objects to build the object graph
- Retrieve named objects from the container

Basic Usage

```
struct A {
    virtual ~A() {}
    virtual int get() = 0;
 };
struct B : A {
    int i;
    B(int i) : i(i) {}
    int get() override { return i; }
};
struct C {
    std::shared_ptr<A> a;
    C(std::shared_ptr<A> a) : a(a) {}
};

Container container;
container.configure(
    type<B>(constructorDependencies(
        valueArgument<int>("i"))).provides<A>(),
    type<C>(constructorDependencies(
        objectArgument<A>("a")))
    );
```

Basic Usage

```
container.configure(
    object("").ofType("B").map("i").to(1),
    object("b2").ofType("B").map("i").to(2),
    object("namedC").ofType("C").map("a").to("b2")
    );

auto defaultC = container.resolve<C>();
assert(defaultC->a->get() == 1);

auto namedC = container.resolve<C>("namedC");
assert(namedC->a->get() == 2);
```

Basic Usage

- For every registered type, there is a default object
- Can override default object configuration by configuring an object with the empty name
- Object names are namespaced to their types

Reflection

- `factory<>` registers factories for types
  - Factories can be arbitrary callables
    - Including member functions of configured objects
  - Specify name of the type or deduce it
  - Specify arguments to the factory
  - Specify which interfaces a type implements
  - Specify optional arguments and initializer function
- `type<>` registers a constructor as a factory

Reflection

- Three kinds of arguments
  - Value arguments, passed by value, created on demand
  - Object arguments, passed by shared_ptr, managed by a lifetime
    - Transient: new object for every request
    - Singleton: one object shared by everyone
    - User-extensible
  - Collection arguments, multiple objects, passed by vector<shared_ptr>

Reflection

- `function` registers a function that can be called in mappings
- `converter` registers a function that converts value arguments

Object Tree

- `object` defines a possibly named object
- Name a type by its registered name
- Bind arguments by their registered names
- Can bind to values specified by users
  - Will try to convert to the right type using converter
- Can bind to other registered objects
- Can bind to inline object mappings
- Can bind to result of function call

Object Tree

- Can select a lifetime for the object
- `alias` defines alternate names for objects
- `decorator` allows injecting additional wrappers into the object tree

Complex Usage

```
struct Command {
  virtual ~Command() {}
  virtual void run() = 0;
};

enum class Action { enter, leave, failed };

struct Logger {
  virtual ~Logger() {}
  virtual void log(std::string_ref sourceType,
                   int sourceId,
                   std::string_ref sourceFunction,
                   Action action) = 0;
};
```

Complex Usage

```
class MultiCommand : public Command {
public:
  MultiCommand(
      std::vector<std::shared_ptr<Command>> commands)
    : commands(std::move(commands))
  {}

  void run() override {
    for (auto& c : commands) {
      c->run();
    }
  }

private:
  std::vector<std::shared_ptr<Command>> commands;
};

container.configure(
  type<MultiCommand>(constructorDependencies(
    collectionArgument<Command>("commands")))
  .provides<Command>());
```

Complex Usage

```
class LoggingCommand : public Command {
public:
  LoggingCommand(std::shared_ptr<Command> inner,
                 std::shared_ptr<Logger> logger,
                 int id)
    : inner(std::move(inner)),
      logger(std::move(logger)), id(id)
  {}

  void run() override {
    try {
      logger->log("Command", id, "run", Action::enter);
      inner->run();
      logger->log("Command", id, "run", Action::leave);
    } catch(...) {
      logger->log("Command", id, "run", Action::failed);
      throw;
    }
  }
```

Complex Usage

```
private:
  std::shared_ptr<Command> inner;
  std::shared_ptr<Logger> logger;
  int id;
};

container.configure(
  type<LoggingCommand>(constructorDependencies(
    objectArgument<Command>("inner"),
    objectArgument<Logger>("logger"),
    valueArgument<int>("id")))
  .provides<Command>());
```

Complex Usage

```
class App {
public:
  App(std::shared_ptr<Command> someCommand)
    : someCommand(std::move(someCommand))
  {}

private:
    std::shared_ptr<Command> someCommand;
};

container.configure(
  type<App>(constructorDependencies(
    objectArgument<Command>("someCommand"))));
```

Complex Usage

```
struct NullLogger : Logger {
  void log(std::string_ref, int, std::string_ref, Action)
      override {}
};

container.configure(type<NullLogger>().provides<Logger>());
```

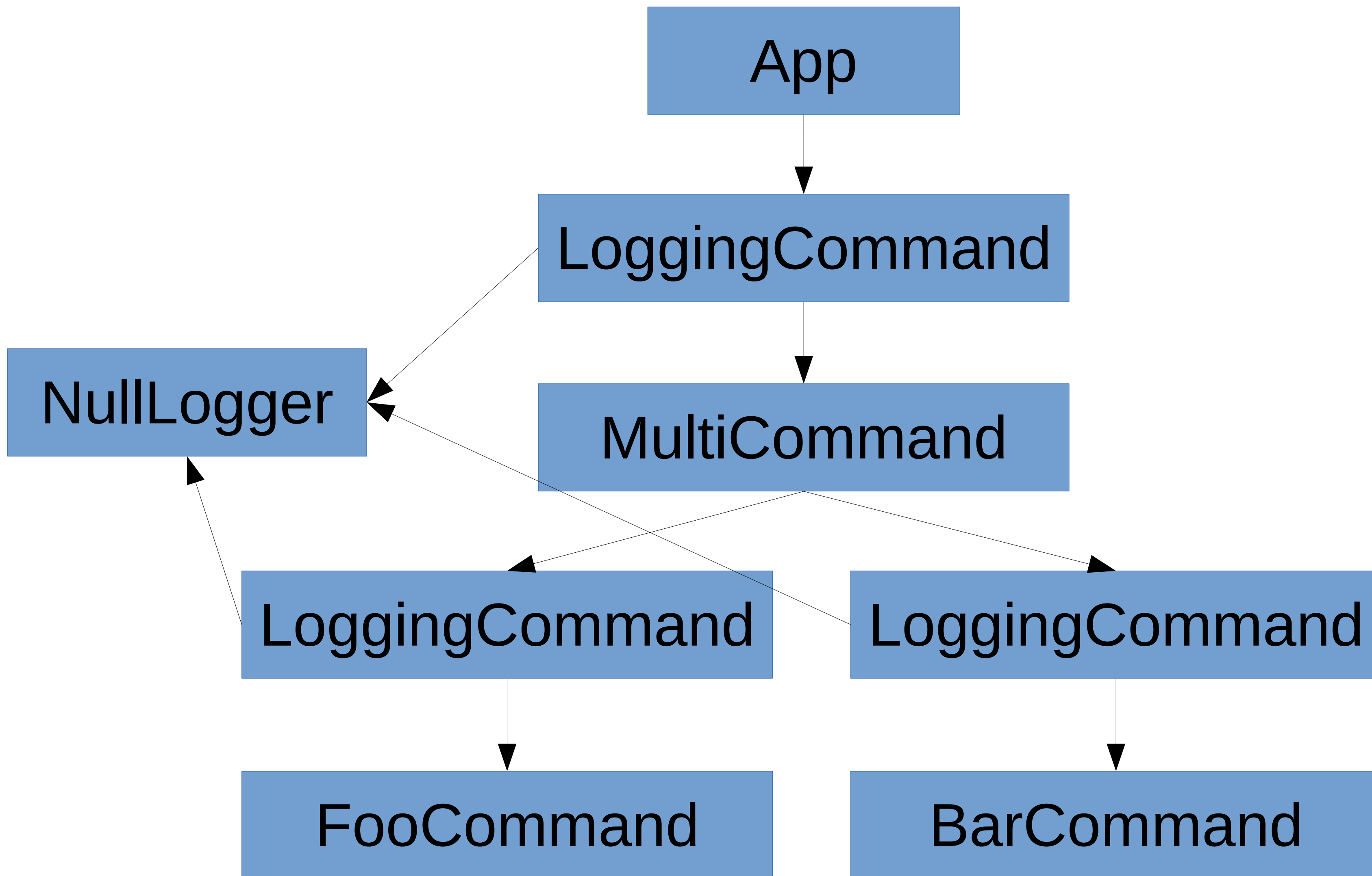
Complex Usage

```
class Sequence {
public:
  Sequence(int initial) : current(initial) {}

  int next() { return current++; }

private:
  int current;
};

container.configure(
  type<Sequence>(constructorDependencies(
    valueArgument<int>("initial").defaultsTo(0))),
  function("nextId", &Sequence::next,
    functionArguments(objectArgument<Sequence>("this"))));
```

Complex Usage

```
container.configure(
  object("combinedCommand").ofType("MultiCommand")
    .map("commands").to({
        sharedObject("FooCommand"),
        sharedObject("BarCommand")
    }),
  decorator("Command").ofType("LoggingCommand")
    .map("inner").to(wrapped())
    .map("logger").to(sharedObject("NullLogger"))
    .map("id").to(
      call("nextId").map("this").to("commandIdSequence")),
  object("commandIdSequence").ofType("Sequence"),
  object("").ofType("App")
    .map("someCommand").to("combinedCommand")
  );

auto app = container.resolve<App>();
```

Complex Usage



Implementation

- Builds graph that describes how to convert values into other values
  - Strings are converted to object configurations by lookup
  - Object configurations are converted to objects by instantiation
- Put in a boost::any and a desired target type
- Get out the modified boost::any or exception

Implementation

- One set of rules to convert values
  - Can be extended by registering converters
  - Allows putting strings from a configuration file directly into mappings and let the container convert to the actual argument type
- One set of rules to convert objects
  - Implicitly extended by registering additional types

Implementation

- A registered object corresponds to a Provider
- Providers can be looked up by name+type pa
- Providers know how to fetch or create (depending on lifetime) their actual object
- Depending on lifetime, one provider can create many instances

Implementation

- A registered type corresponds to a ProviderFactory
- A factory can be looked up by the dynamic typ name
- The object declaration looks up the factory and uses it to create a provider

Disadvantages

- Generates a lot of code
- Lots of overhead on object creation
- Configuration errors only detected at runtime
  - Except for wrong reflection information
- Bound to using shared_ptr for objects
- Not thread-safe at all

Future Directions

- Make thread-safe
- Possibly replace shared_ptr with a Handle type that can deal with dependencies on weird lifetimes
  - For example, a singleton object depending on a session-scoped object
- Make open-source
  - I need you to tell me if this would be useful

Questions?