C++Now 2016

C++14 DEPENDENCY INJECTION LIBRARY

https://github.com/boost-experimental/di

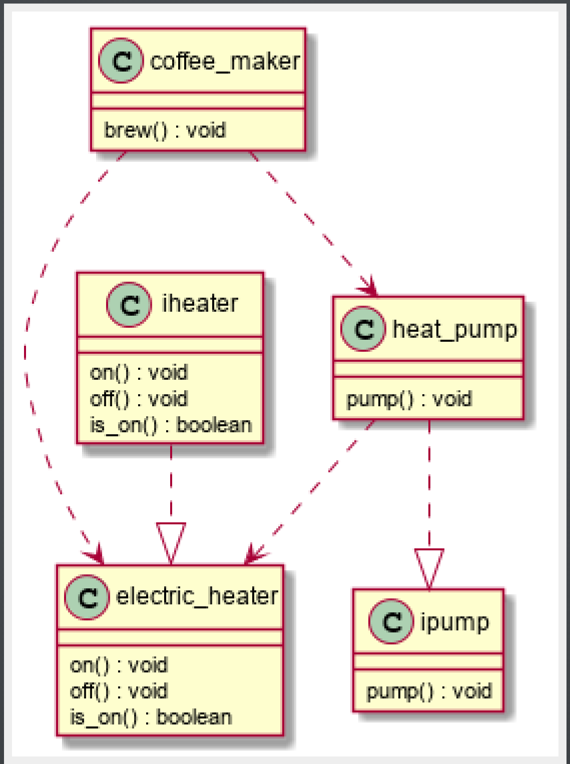
Kris Jusiak

DEPENDENCY INJECTION

(DI) involves passing (injecting) one or more dependencies (or services) to a dependent object (or client) which become part of the client’s state. It is like the Strategy Pattern, except the strategy is set once, at construction. DI enables loosely coupled designs, which are easier to maintain and test

Rob Stewart

"LET'S MAKE SOME COFFEE!"



NO DEPENDENCY INJECTION

```
class coffee_maker {
public:
  // create dependencies in the constructor
  coffee_maker()
    : heater(std::make_shared<electric_heater>())
    , pump(std::make_unique<heat_pump>(heater))
  { }

  void brew() {
    heater->on();
    pump->pump();
  }

private:
  std::shared_ptr<iheater> heater;
  std::unique_ptr<ipump> pump;
};
```

DEPENDENCY INJECTION

```
class coffee_maker {
public:
  // inject dependencies via constructor
  coffee_maker(std::shared_ptr<iheater> heater
             , std::unique_ptr<ipump> pump)
    : heater(heater), pump(std::move(pump))
  { }

  void brew() {
    heater->on();
    pump->pump();
  }

private:
  std::shared_ptr<iheater> heater;
  std::unique_ptr<ipump> pump;
};
```

IT'S ALL ABOUT THE CONSTRUCTION!

"Don't call us, we'll call you", Hollywood principle

DO I NEED DEPENDENCY INJECTION?

NO, BUT...

- DI promote loosely coupled code
  - Separation of business logic and object creation
  - Expresses `WHAT`, not `HOW`!
- DI creates easier to maintain code
  - Simplified refactoring of dependencies
- DI creates easy to test code
  - Fakes objects might be injected (automatically)

DO I NEED A DI FRAMEWORK/LIBRARY?

NO, BUT DI LIBRARY WILL **FREE YOU** FROM MAINTAINING BOILERPLATE CODE

```
auto create() {
  logger logger_;
  renderer renderer_;
  view view_{renderer_, logger_};
  model model_{logger_};
  controller controller_{model_, view_, logger_};
  user user_{logger_};
  ...
  return make_unique<app>(controller, user_, logger_).run();
}
```

Boilerplate code which has to be maintained

- ORDER in which above dependencies are created is **IMPORTANT**
- **ANY** change in **ANY** of the objects constructor will **REQUIRE** a change in the code

SHOWCASE / MOTIVATION

http://melpon.org/wandbox/permlink/m103wMvJYyRhDdkU

DI CAN ALSO HELP WITH

- Testing
  - Mocks provider
- Serializing
  - Automatic serialization of PODs
- Understanding dependencies
  - Dump relationship between types
- Restricting allowed types
  - Disallow raw pointers, etc.

TRY IT YOURSELF ONLINE!

http://boost-experimental.github.io/di/try_it

STILL NOT CONVINCED?

REAL-LIFE EXAMPLE?

Let's make a web match-3 game in C++14

- Emscripten
- Ranges
- Dependency Injection
- Meta State Machine

---

https://github.com/modern-cpp-examples/match3

C++ VS JAVA VS C# LIBRARIES

WRITING A DI LIBRARY IS NOT EASY

In C++ it's even harder

- Performance is important
- Lack of static reflection
- Pointers, References, Rvalues, Smart Pointers, ...
- Qualifiers - const, volatile, ...
- Templates, Concepts, ...

DI LIBRARIES

Library	Boost.DI	Google.Fruit	Google.Guice	Dagger2	Ninject
Language	C++14	C++11	Java	Java	C#
Version	1.0.0	2.0.2	4.0	2.4	3.2
License	Boost 1.0	Apache 2.0	Apache 2.0	Apache 2.0	Apache 2.0
Linkage	header only	library	jar	jar	dll
Approach	compile-time	compile/run-time	run-time	annotation processor	run-time
Errors	compile-time	compile-time + exceptions	exceptions	compile-time	exceptions

BENCHMARKS

CREATE UNIQUE OBJECTS TREE

BASELINE - OBJECTS CREATED MANUALLY

Types = 64 | Constructor parameters <= 4

	Baseline	Boost.DI	Google.Fruit	Google.Guice	Dagger2	Ninject
Compilation time	0.063s	0.376s	2.329s	0.570s	1.411s	0.144s + 0.079s
Executable size	4.2K	8.5K	213K	-	-	-
Execution time	0.002s	0.002s	0.037s	0.528s	0.157s	1.131s

Types = 256 | Constructor parameters <= 4

	Baseline	Boost.DI	Google.Fruit	Google.Guice	Dagger2	Ninject
Compilation time	0.131s	1.328s	9.641s	0.783s	2.814s	0.151s + 0.114s
Executable size	4.2K	8.7K	1.4M	-	-	-
Execution time	0.003s	0.003s	0.154s	0.723s	0.323s	4.838s

Types = 512 | Constructor parameters <= 4

	Baseline	Boost.DI	Google.Fruit	Google.Guice	Dagger2	Ninject
Compilation time	0.215s	2.459s	23.924s	1.054s	4.231s	0.157s + 0.161
Executable size	8.2K	13K	4.2M	-	-	-
Execution time	0.003s	0.003s	0.328s	0.943s	0.547s	11.123s

MORE BENCHMARKS

http://boost-experimental.github.io/di/benchmarks

EXPERIMENTAL BOOST.DI

OVERVIEW

A BIT OF HISTORY

2012 - 2014

Version C++98 / C++11 - never released

https://github.com/boost-experimental/di/tree/cpp03

- Compiled slowly (Boost.MPL)
- Long error messages
- A lot of preprocessor magic (BOOST_PP)
- A lot of workarounds for compilers (MSVC 2013)

2014 - Now

Version C++14 - v1.0.1

---

https://github.com/boost-experimental/di

- One header (boost/di.hpp) / generated
- 3k lines
- Neither Boost nor STL is required
- No 'if' branches
- No 'virtual' methods
- No 'exceptions' (-fno-exceptions)

TESTED COMPILERS

- Clang-3.4+
- XCode-6.1+
- GCC-5.2+
- MSVC-2015+

QUALITY (PER COMMIT)

CONTINOUS INTEGRATION

- Build with `-Wall -Wextra -Werror -pedantic -pedantic-errors`
- Travis/Appveyor build on Linux/OS X/Windows (`Boost-Build/CMake`)
    - Clang-3.4/3.5/3.6/3.7/3.8 (`libc++/stdlibc++`)
    - GCC-5
    - MSVC-2015
- Clang static analysis / Clang-tidy (static check)
- Valgrind / Dr. Memory (dynamic memory check)
- Clang-format (style check)
- Documentation deployment to 'GitHub/gh-pages'

TESTS

99% TEST CODE COVERAGE

+

101 EXAMPLES

- Unit tests
- Functional tests
- Performance tests / Benchmarks
- Compilation-error tests (Validates error message)
- Run-time performance tests (Compares generated assembler opcodes)

DESIGN

GOALS

- BE AS FAST AS POSSIBLE
    - Boost.DI has none or minimal run-time overhead

- COMPILE AS FAST AS POSSIBLE
  - Boost.DI compiles **faster than Java-Dagger2!**

- ## GUARANTEE OBJECT CREATION AT COMPILE-TIME
  - Boost.DI resolves types at compile-time and gives short and intuitive error messages

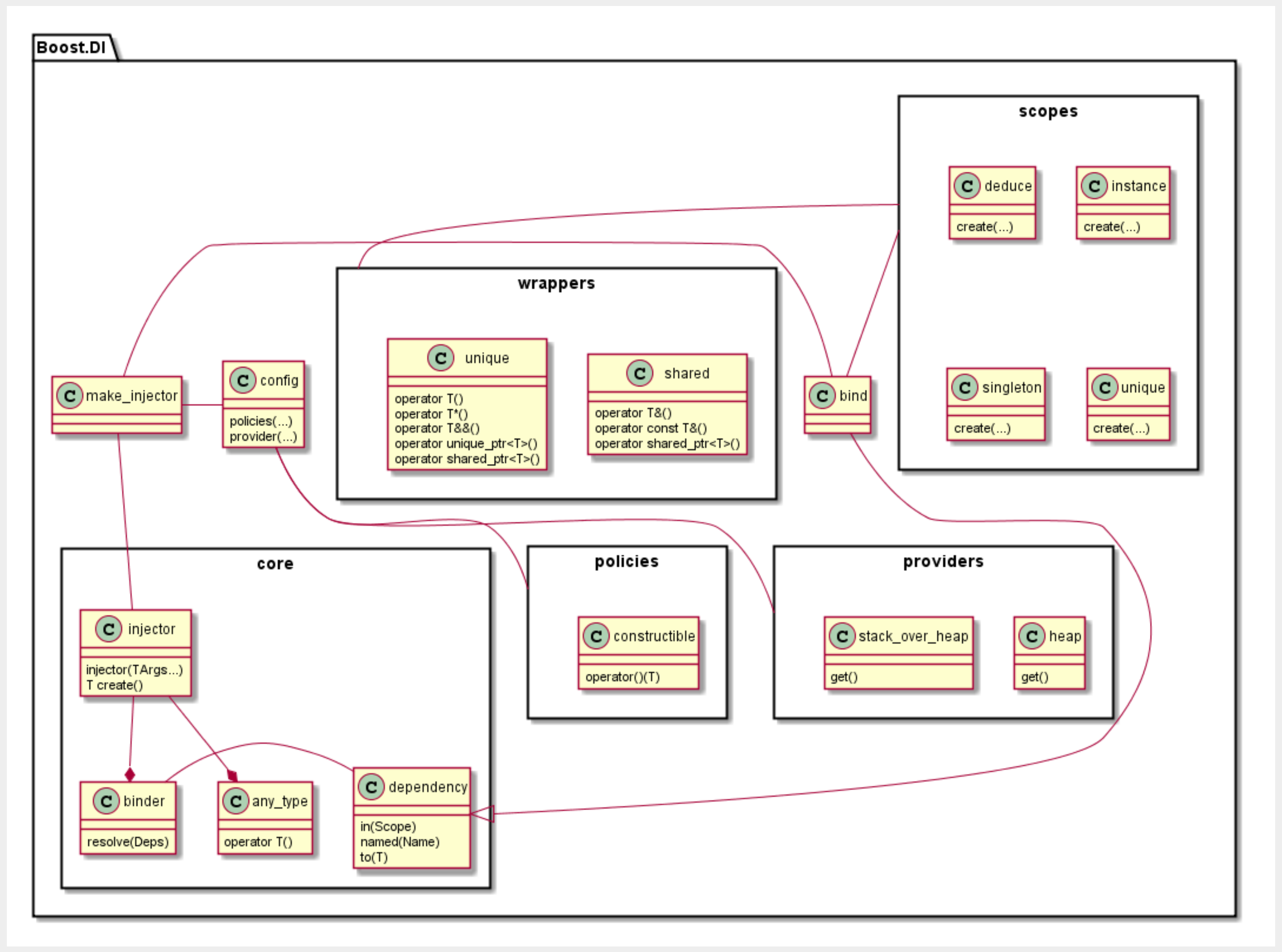
If it compiles it will work!

- BE AS NON-INTRUSIVE AS POSSIBLE
    - Boost.DI deduces constructor parameters without reflection

- BE EASY TO EXTEND
  - Boost.DI provides easy way to write custom scopes/policies/providers

ARCHITECTURE

DESIGN


Boost.DI
scopes
deduce
create(...)
instance
create(...)
singleton
create(...)
unique
create(...)
wrappers
unique
operator T()
operator T*()
operator T&&()
operator unique_ptr<T>()
operator shared_ptr<T>()
shared
operator T&()
operator const T&()
operator shared_ptr<T>()
make_injector
config
policies(...)
provider(...)
bind
core
injector
injector(TArgs...)
T create()
binder
resolve(Deps)
any_type
operator T()
dependency
in(Scope)
named(Name)
to(T)
policies
constructible
operator()(T)
providers
stack_over_heap
get()
heap
get()

- Bindings
    - DSL to create dependencies representation which will be used by core to resolve types

- Scopes
    - Responsible for maintain objects life time

- Providers
    - Responsible for providing object instance

- Policies
    - Compile-time limitations for types / Run-time types visitor

- Config
  - Configuration for Policies and Providers

IN A NUTSHELL (PSEUDO-CODE)

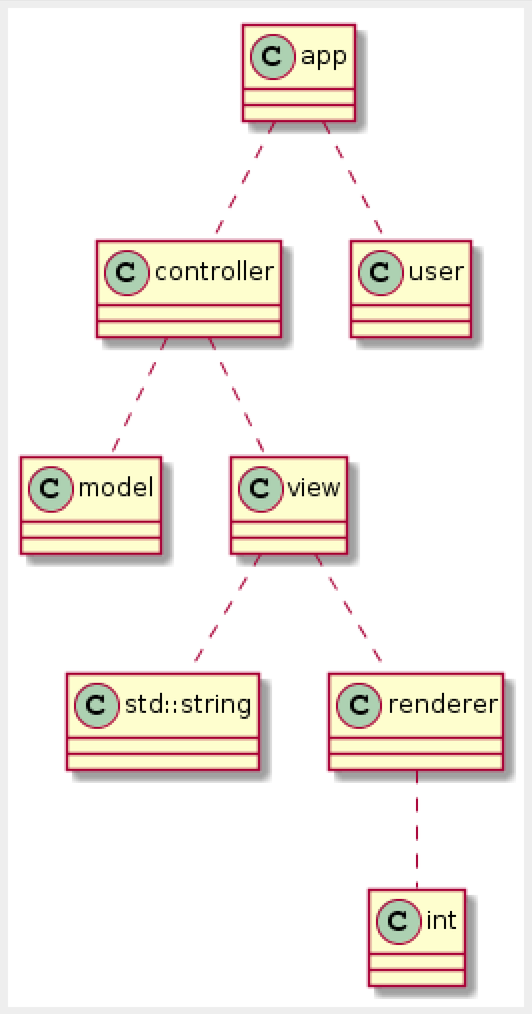
DESIGN

```cpp
template<class TConfig, class... TBindings>
class core::injector {
  template<class T> constexpr auto create() const noexcept {
    TConfig::policies<T>()...;

    auto&& dependency = binder{}.resolve<T>(TBindings...);
    using ctor = ctor_traits<injector, T>();

    return wrapper<T>{
      dependency.create( // create in a dependency scope
        TConfig::provider{}.get<decltype(dependency.impl)>(
          create<ctor>()...))
    };
  }
};
```

USER GUIDE

CREATE OBJECTS TREE

APP



APP

```cpp
struct renderer { int device; };
class iview {
public:
  virtual ~iview() = default;
  virtual void update() = 0;
};
class model {};
class controller {
public:
  controller(model&, view&) {}
};
class user {};
class app {
public:
  app(controller&, user&) {}
};
```

USUAL APPROACH TO CREATE `APP`

CREATE APP

```
renderer renderer_;
view view_{"title", renderer_};
model model_;
controller controller_{model_, view_};
user user_;
app app_{controller_, user_};
```

WITH BOOST.DI

CREATE APP

```cpp
auto app = di::make_injector().create<app>();
```

WHERE

MAKE INJECTOR

```
template<class TConfig = di::config, class... TDeps>
  requires configurable<TConfig>() && boundable<TDeps...>()
auto make_injector(TDeps&&...) noexcept;
```

INJECTOR

```
template<class... TDeps> requires boundable<TDeps...>()
class injector {
public:
  explicit injector(TDeps&&...) noexcept;
  injector(injector&&) = default;
  injector& operator=(injector&&) = default;

  template<class T> requires creatable<T>()
  constexpr T create() const;
};
```

HOW IS THAT POSSIBLE WITHOUT STATIC REFLECTION?

USER-DEFINED/IMPLICIT/GENERIC CONVERSION OPERATOR

ANY TYPE

```cpp
struct any_type {
  template<class T>
  constexpr operator T(); // non explicit
};

int main() {
  struct example {
    example(int, double);
  };

  static_assert(
    std::is_constructible<example, any_type, any_type>::value);
}
```

http://melpon.org/wandbox/permlink/55bToJVYIWO4gald

PROBLEM - COPY CONSTRUCTOR / MOVE CONSTRUCTOR

```
static_assert(
  !std::is_constructible<example, any_type>::value;
);
```

http://melpon.org/wandbox/permlink/KFYmTrdJpTjB6UEr

SOLUTION

Disable the operator when type T is convertible to the parent type

ANY TYPE V2

```cpp
template<class TParent>
struct any_type {
  template<class T, class =
    std::enable_if_t<!std::is_convertible<TParent, T>{}>
  > constexpr operator T();
};

int main() {
  struct example {
    example(int, double);
  };

  static_assert(
    !std::is_constructible<example, any_type<example>>::value);
}
```

http://melpon.org/wandbox/permlink/v7OIgdzA81TtVF5a

GENERIC CONVERTING CONSTRUCTOR?

EXAMPLE

```
class example {
public:
  template<class T>
  example(T); // non explicit
};
```

SOLUTION

- Restrict allowed types `T`
- Register constructor explicitly via [inject]

STD.FUNCTION

```
template<typename _Res, typename... _ArgTypes>
class function<_Res(_ArgTypes...)> {
public:
  template<typename _Functor, typename =
    _Requires<_Callable<_Functor>, void>> // solve the issue
    function(_Functor);
};
```

CALCULATE THE NUMBER OF PARAMETERS?

IS CONSTRUCTIBLE

```cpp
constexpr auto BOOST_DI_CFG_CTOR_LIMIT_SIZE = 10;

template<class T, std::size_t>
using any_type_t = any_type<T>;

template<class...>
struct is_constructible;

template<class T, std::size_t... Ns>
struct is_constructible<T, std::index_sequence<Ns...>>
  : std::is_constructible<T, any_type_t<T, Ns>...>
{ };
```

NUMBER OF CONSTRUCTOR PARAMETERS

```
template <class T, std::size_t... Ns>
constexpr auto get_ctor_size(std::index_sequence<Ns...>) noexcept {
  auto value = 0;
  int _[]{0, (is_constructible<T, std::make_index_sequence<Ns>>{}
    ? value = Ns : value)...};
  return value;
}

int main() {
  struct example {
    example(int, double, float);
  };

  static_assert(3 == get_ctor_size<example>(
    std::make_index_sequence<BOOST_DI_CFG_CTOR_LIMIT_SIZE>{}));
}
```

http://melpon.org/wandbox/permlink/xoKrb40GYTi5deoJ

HOW IT'S DONE IN DI?

IS BRACES CONSTRUCTIBLE

IS BRACES CONSTRUCTIBLE

```
template <class T, class... TArgs>
decltype(void(T{declval<TArgs>()...}), true_type{})
test_is_braces_constructible(int);

template <class, class...>
false_type test_is_braces_constructible(...);

template <class T, class... TArgs>
using is_braces_constructible =
  decltype(test_is_braces_constructible<T, TArgs...>(0));

template <class T, class... TArgs>
using is_braces_constructible_t =
  typename is_braces_constructible<T, TArgs...>::type;
```

```
struct example { int a; int b; };
static_assert(is_braces_constructible<example, any_type, any_type>{});
```

IMPLEMENTATION (PSEUDO-CODE)

ANY TYPE V3

```
template<class TInjector, class TParent>
struct any_type {
  template<class T, class =
    std::enable_if_t<!std::is_convertible<TParent, T>{}>
    ... // Concepts
  > constexpr operator T() {
    return injector_.template create<T>();
  }

  const TInjector& injector_;
};
```

CONSTRUCTOR TRAITS

```
template<class TInjector, class T> auto ctor_traits() {
  if (has_inject<T>() { // BOOST_DI_INJECT
    return pair<direct, typename T::inject>{};
  }
  for (auto i = BOOST_DI_CFG_CTOR_LIMIT_SIZE; i >= 0; --i) {
    if (is_constructible<T, any_type<TInjector, T>...>())
      return pair<direct, any_type<TInjector, T>...>{};
    }
  }
  for (auto i = BOOST_DI_CFG_CTOR_LIMIT_SIZE; i >= 0; --i) {
    if (is_braces_constructible<T, any_type<TInjector, T>...>())
      return pair<uniform, any_type<TInjector, T>...>{};
    }
  }
  return error(...); // concepts emulation
};
```

COMING BACK TO THE DESIGN

DESIGN

```
template<class TConfig, class... TBindings>
class core::injector {
  template<class T> constexpr auto create() const noexcept {
    TConfig::policies<T>()...;
    auto&& dependency = binder{}.resolve<T>(TBindings...);

    using ctor = ctor_traits<injector, T>(); // -\
     // pair<direct/uniform, TCtor...>  <--------/

    return wrapper<T>{
      dependency.create( // create in a dependency scope
        TConfig::provider{}.get<decltype(dependency.impl)>(
          create<ctor>()...))
    };
  }
};
```

BINDINGS

DI Configuration

INTERFACES

INTERFACE -> IMPLEMENTATIONS

```cpp
class iview {
public:
  virtual ~iview() noexcept = default;
  virtual void update() =0;
};

class gui_view: public iview {
public:
  gui_view(std::string title, const renderer&) {}
  void update() override {}
};

class text_view: public iview {
public:
  void update() override {}
};
```

BINDINGS

```
auto injector = di::make_injector(
  di::bind<iview>.to<gui_view>() // bind interface to implementation
);
```

TEST

```
assert(dynamic_cast<gui_view*>(
  injector.create<std::unique_ptr<iview>().get())
);
```

VALUES

AGGREGATE

```
struct T { // create using uniform initialization
  int& a;  // might be used to serialize
  double b;
};
```

BINDINGS

```
 auto i = 42;
 auto injector = di::make_injector(
    di::bind<int>.to(i),
    di::bind<double>.to(87.0)
 );
 injector.create<T>(); // will create T{i, 87.0};
```

DYNAMIC CONDITIONS

BINDINGS

```
auto use_gui_view = true/false;

auto injector = di::make_injector(
  di::bind<iview>.to([&](const auto& injector) -> iview& {
    return use_gui_view ?
      injector.template create<gui_view&>() :
      injector.template create<text_view&>();
  })
);
```

TEST

```
use_gui_view = true;
assert(dynamic_cast<gui_view*>(
  injector.create<std::unique_ptr<iview>().get())
);

use_gui_view = false;
assert(dynamic_cast<text_view*>(
  injector.create<std::unique_ptr<iview>().get())
);
```

THIS WAY XML INJECTION MIGHT BE EASILY ACHIEVED

XML Injection

VECTORS/LISTS/ARRAYS/...

USING INITIALIZER LIST

BINDINGS

```
auto injector = di::make_injector(
  di::bind<int[]>().to({1, 2, 3})
);
```

TEST

```
auto v = injector.create<std::vector<int>>();
         // or std::array / std::set

assert(3 == v.size());
assert(1 == v[0]);
assert(2 == v[1]);
assert(3 == v[2]);
```

USING LIST OF TYPES

BINDINGS

```
auto injector = di::make_injector(
  di::bind<interface*[]>().to<implementation1, implementation2>()
);
```

TEST

```
auto v = injector.create<
  std::vector<std::unique_ptr<interface>>>();
    // or std::array / std::set with
    // std::shared_ptr, raw pointer, reference, ...

assert(2 == v.size());
assert(dynamic_cast<implementation1*>(v[0].get()));
assert(dynamic_cast<implementation2*>(v[1].get()));
```

SCOPES

Objects life time

- Deduce scope (default)
- Instance scope (bind<>.to(value) where value is maintained by the user)
- Unique scope (one instance per request)
- Singleton scope (shared instance)

SCOPES LIFE TIME

```
+----------------------------------------------+ Singleton scope
|
|          Unique scope     Unique scope
|          +---+            +------+
|          |                |
+--+-------+----------------+------------------> Application life time
   |
   |
   +------------------------+ Instance / User scope
                             (is not controlled by DI)
```

SCOPES DEDUCTION

Type	Scope
`T`, `T&&`, `T*`, `const T*`, `std::unique_ptr<T>`	Unique scope
`T&`, `const T&`, `std::shared_ptr<T>`, `boost::shared_ptr<T>`, `std::weak_ptr<T>`	Singleton scope

EXPLICIT CHANGE OF THE SCOPE FOR A GIVEN TYPE

BINDINGS

```cpp
auto injector = di::make_injector(
  di::bind<iview>.to<gui_view>().in(di::singleton) // explicitly
);
```

TEST

```cpp
assert(&injector.create<iview&>() == &injector.create<iview&>());
```

INJECTIONS / ANNOTATIONS

AMBIGUOUS CONSTRUCTORS

```
class model {
 public:
   model(int size, double precision) { }
   model(int rows, int cols) { }
};
```

Constructor ambiguity (compilation error)

SOLUTION

INJECT

```
class model {
 public:
   model(int size, double precision) { assert(false); }
   BOOST_DI_INJECT(model, int rows, int cols); // pick me!
};

model::model(int rows, int cols) {} // implementation is not affected
```

TEST

```
auto injector = di::make_injector();
injector.create<model>(); // compiles and run
```

DISTINGUISH ROWS FROM COLUMNS

ANNOTATIONS / NAMES

NAMES

```
class model {
 public:
   model(int size, double precision) { }

   BOOST_DI_INJECT(model, (named = "rows"_s) int rows
                        , (named = "cols"_s) int cols);
};

model::model(int rows, int cols) {} // implementation stays the same
```

BINDINGS

```
auto injector = di::make_injector(
  di::bind<int>.named("rows"_s).to(6)
, di::bind<int>.named("cols"_s).to(8)
);
```

WHERE

COMPILE-TIME STRING

```
template <char...>
struct string {};

template <class T, T... Ts>
constexpr auto operator""_s() {
  return string<Ts...>{}
}
```

It's not standard! For a standard solution use unique types instead

```
auto rows = []{}; // using r = decltype(rows);
auto cols = []{}; // static_assert(!std::is_same<decltype(cols), r>{});
```

MODULES

Split DI configuration

MODULES

```
auto view_module = [] {
  return di::make_injector(
    di::bind<icanvas>.to<sdl_canvas>()
  , di::bind<irenderer>.to<gui_renderer>()
  );
};

auto model_module = [] {
  return di::make_injector(
    di::bind<config>.to({6, 8})
  , di::bind<irandom>.to<mt19937_random>()
  );
};
```

BINDINGS

```
auto injector = di::make_injector(
  view_module(), model_module()
);

injector.create<app>();
```

MODULE IN CPP FILE

EXPOSE TYPES VIA INJECTOR

Only exposed types will be creatable

EXPOSE MODULES

```
di::injector<view&> view_module() { // expose view
  return di::make_injector(
    di::bind<icanvas>.to<sdl_canvas>()
  , di::bind<irenderer>.to<gui_renderer>()
  );
}
```

```
di::injector<model&> model_module() { // expose model
  return di::make_injector(
    di::bind<config>.to({6, 8})
  , di::bind<irandom>.to<mt19937_random>()
  );
}
```

BINDINGS

```
auto injector = di::make_injector(
  view_module(), model_module()
);

injector.create<app>();
```

ADDITIONAL READINGS

http://boost-experimental.github.io/di/user_guide

http://boost-experimental.github.io/di/tutorial

ERROR MESSAGES / CONCEPTS

CONCEPTS EMULATION

COMMON APPROACH WITHOUT CONCEPTS

REQUIRES

```
#define REQUIRES(...) \
  typename std::enable_if<__VA_ARGS__, int>::type = 0
```

EXAMPLE

```
template<class T, REQUIRES(std::is_same<T, int>{})>
void call_if_int();
```

EXAMPLE - PROVIDABLE

```
template <class...>
using is_valid_expr = true_type;
```

```
template <class T>
std::false_type providable_impl(...); // or some type error message

template <class T>
auto providable_impl(T&& t) -> is_valid_expr<
  decltype(t.template is_creatable<T>())
, decltype(t.template get<T>())
>;

template <class T>
constexpr auto providable() {
  return decltype(providable_impl(std::declval<T>()))::value;
}
```

HOW TO GET BETTER ERROR MESSAGES?

TRANSPORT THE SUBSTITUTION FAILURES

PROBLEM

User-defined/implicit/generic conversion operator

ANY TYPE

```
struct any_type {
  template<class T>
  constexpr operator T(); // no easy way to return a failure from T
};
```

SOLUTION

SPLIT CONCEPTS EMULATION INTO 2 PARTS

CHECK THE PREDICATE

DISPLAY THE ERROR MESSAGE

HOW?

CREATABLE / CONCEPT

CREATABLE

```cpp
template <class T, class TDependency = binder::resolve_t<T>>
using creatable = std::is_convertible<
  TDependency::template try_create<T>(
    provider<ctor_traits<typename TDependency::impl>>{}
  );
>;
```

WHERE

```cpp
template <class T, class TProvider>
auto try_create(const TProvider& provider) -> wrapper<
  unique, decltype(provider.get())> // `auto -> decltype` will disable
                                    // function if not applicable
try_create(const TProvider&);
```

CONSTRUCTOR DEDUCTION

```
struct any_type {
  template <class T, REQUIRES(creatable<T>())>
  constexpr operator T(); // disabled when type is not creatable
};
```

SHOW THE ERROR

ERROR DEDUCTION / SIMPLIFIED

```
template<class T>
constexpr auto show_the_error() {
  return aux::is_polymorphic<T>{} ?
    abstract_type<T>::is_not_bound{} :
    type<T>::cant_be_created{};
};
```

Static inline function without implementation will show a warning without a CALL STACK!

ABSTRACT TYPE IS NOT BOUND

ABSTRACT TYPE IS NOT BOUND

```cpp
template <class T>
struct abstract_type {
  struct is_not_bound {
    constexpr operator T() const { return error(); }

    // no implementation
    static inline T
      error(_ = "type is not bound, did you forget to add:
          'di::bind<interface>.to<implementation>()'?");
  };
};
```

CHANGE THE WARNING INTO ERROR

LIFT WARNINGS INTO ERRORS

```
#if defined(__clang__)
#pragma clang diagnostic error "-Wundefined-inline"
#elif defined(__GCC__)
#pragma GCC diagnostic error "-Werror"
#elif defined(__MSVC__)
#pragma warning(disable : 4822)
#endif
```

FROM TOP TO BOTTOM

INJECTOR

```
template <class T, REQUIRES(creatable<T>())>
constexpr T create() const {
  return create_successful_impl(type<T>{}); // compilation time speed up
}

template <class T, REQUIRES(!creatable<T>())>
[[deprecated("creatable constraint not satisfied")]]
constexpr T create() const {
  return create_impl(type<T>{});
}
```

ERROR DEDUCTION

```
template<class T, class... TArgs,
  REQUIRES(std::is_constructible<T, TArgs...>{})>
  // TArgs migth be disabled by any_type
constexpr auto create_impl(TArgs&&... args) {
  return T{std::forward<TArgs>(args...)};
}

template<class T, class... TArgs,
  REQUIRES(!std::is_constructible<T, TArgs...>{})>
constexpr auto create_impl(TArgs&&... args) {
  return show_the_error<T>{};
}
```

EXAMPLE

APP

```
struct renderer {
  int device;
};
class view {
public:
  view(std::string title, const renderer&);
};
class model {};
class controller {
public:
  controller(model&, iview&) {} // iview interface
};
class user {};
class app {
public:
  app(controller&, user&) {}
};
```

```
auto injector = di::make_injector(); // no bindings for iview
injector.create<app>(); // compilation error
```

COMPILATION ERROR MESSAGE

CLANG

```
error: 'create<app>' is deprecated: creatable constraint not satisfied
  injector.create<app>();
           ^
note 'create<app>' has been explicitly marked deprecated here
  create
  ^
error: inline function 'abstract_type<iview>::is_not_bound::error'
 error(_ = "type is not bound, did you forget to add:
           'di::bind<interface>.to<implementation>()'?");
```

GCC

```
error: ‘T injector<...>::create() const [with T = app]’ is deprecated:
  creatable constraint not satisfied
   injector.create<app>();
                        ^
note declared here
   create
   ^
error: inline function ‘abstract_type<T>::is_not_bound::error(_)
  [with T = iview]’ used but never defined
  error(_ = "type is not bound, did you forget to add:
            'di::bind<interface>.to<implementation>()'?");
```

MSVC

```
error C4996: 'injector<...>::create': creatable constraint not
                                      satisfied
        with
        [
            TConfig=config
        ]
note see declaration of 'injector<...>::create'
        with
        [
            TConfig=config
        ]
error C4506: no definition for inline function
            'abstract_type<T>::is_not_bound::error(_)'
        with
        [
            T=iview
        ]
```

Suggestions are not shown on MSVC

BOOST DI CFG DIAGNOSTICS LEVEL

BOOST DI CFG DIAGNOSTICS LEVEL=0

```
error: 'create<app>' is deprecated: creatable constraint not satisfied
```

BOOST DI CFG DIAGNOSTICS LEVEL=1

Default

```
error: 'create<app>' is deprecated: creatable constraint not satisfied
  injector.create<app>();

error: inline function 'abstract_type<iview>::is_not_bound::error'
 error(_ = "type is not bound, did you forget to add:
           'di::bind<interface>.to<implementation>()'?");
```

BOOST DI CFG DIAGNOSTICS LEVEL=2

```
error: 'create<app>' is deprecated: creatable constraint not satisfied

error: function ‘T creating<T>::type(_) [with T = app]’
error: function ‘T creating<T>::type(_) [with T = controller]’

error: inline function 'abstract_type<iview>::is_not_bound::error'
 error(_ = "type is not bound, did you forget to add:
           'di::bind<interface>.to<implementation>()'?");
```

EXTENSIONS

PROVIDE AN EASY WAY TO EXTEND DI FUNCTIONALITY WITHOUT CHANGING THE CORE

- Scopes
  - Customize object life time
- Policies
  - Check/Visit created objects
- Providers
  - Customize object creation

PRINT TYPES / POLICY

DUMP TYPES POLICY

```
struct types_dumper : di::config {
  static auto policies(...) noexcept {
    return di::make_policies([](auto type) {
      using T = decltype(type);
      using ctor = typename T::type;
      using impl = typename T::given;
      std::cout << ... << std::endl;
    });
  }
};
```

EXAMPLE OUTPUT

APP

```cpp
class iview {
public:
  virtual ~iview() noexcept = default;
  virtual void update() =0;
};
struct model { std::vector<int> board; };
class controller {
public:
  controller(model&, iview&) {}
};
struct user {};

class app {
public:
  app(controller&, user&) {}
};
```

TYPES DUMPER

```
auto injector = di::make_injector<types_dumper>(
  di::bind<iview>.to<gui_view>()
);

injector.create<app>();
```

- app
  - controller
    - model
      - int[]
    - iview -> gui_view
  - user

SERIALIZE / POLICY

PODs only

SERIALIZABLE

CONFIGURATION

```
struct serializable : di::config {
  template <class TInjector>
  static auto policies(const TInjector& injector) noexcept {
    return di::make_policies([&](auto type) {
      using T = decltype(type);
      ...
      auto& serialize = injector.template create<serializable&>();
      auto ptr = reinterpret_cast<char*>(&injector.template create<T&>()
      const auto offset = calculate_offset(sizeof(T), alignof(T));
      serialize.emplace_back({get_type<T>(), ptr, offset});
      ...
    }
  }
};
```

SERIALIZE

SERIALIZE CLOSURE

```
auto serialize = [](const auto& injector, auto& str) {
  serializable_call_t::apply(injector, [&](const auto& o, auto t) {
    str << o.path << " "
        << o.type << " "
        << o.offset << " "
        << std::to_string(
             *reinterpret_cast<decltype(t)*>(o.ptr() + o.offset)
           )
        << std::endl;
  });
};
```

DESERIALIZE

DESERIALIZE CLOSURE

```
auto deserialize = [](const auto& injector, auto& str) {
  serializable_call_t::apply(
    injector, [&](const auto& o, auto t, auto line) {
    std::string line, path, type;
    decltype(t) value = {};
    auto offset = 0;
    std::istringstream iss{line};
    iss >> path >> type >> offset >> value;
    *reinterpret_cast<decltype(t)*>(o.ptr() + offset) = value;
  });
};
```

EXAMPLE

APP

```
struct data { unsigned int ui; long l; float f; };
struct even_more_data { double d; bool b; long long ll; };
struct more_data { int i; long double ld; even_more_data d; short s; };

class app {
public:
  app(data& d, more_data& md) : d(d), md(md) {}

  void update(); // change data, more_data

private:
  data& d;
  more_data& md;
};
```

CREATE INJECTOR

```
auto injector = di::make_injector<serializable>();
injector.create<app>();
```

SERIALIZE

```
std::stringstream str;
serialize(injector, str);
```

DESERIALIZE

```
deserialize(injector, str);
```

EXAMPLE OUTPUT

```
app->data unsigned_int 13
app->data long 23
app->data float 0.330000
app->more_data int 44
app->more_data long_double 42.000000
app->more_data->even_more_data double 55.000000
app->more_data->even_more_data bool 1
app->more_data->even_more_data long_long 66
app->more_data short 77
```

CONSTRUCTIBLE / POLICY

LET'S DISALLOW TYPES WHICH ARE NOT PODS OR ARE NOT BOUND

CONFIGURATION

```
struct is_pod_or_is_bound : di::config {
  static auto policies(...) noexcept {
    using namespace di::policies;
    return di::make_policies(
      constructible(std::is_pod<_>{} || is_bound<_>{})
    );
  }
};
```

APP

```cpp
struct not_a_pod { virtual ~not_a_pod() = default; };
struct app { app(not_a_pod, int, double) { } };
```

INJECTOR

```cpp
auto injector = di::make_injector<is_pod_or_is_bound>(
  di::bind<>().to(42)
, di::bind<not_a_pod>().to(not_a_pod{})
);

injector.create<app>();
```

ERROR CASE / COMPILATION ERROR

CREATE

```
di::make_injector<is_pod_or_is_bound>().create<app>();
```

ERROR MESSAGE

```
error: 'create<app>' is deprecated: creatable constraint not satisfied
  injector.create<app>();
           ^
error: inline function 'type<not_a_pod>::not_allowed_by<
  or_<std::is_pod<_>, is_bound<_>>>::error'

  error(_ = "type disabled by constructible policy
           , added by BOOST_DI_CFG or make_injector<CONFIG>!");
```

Error will be shown for ALL types which don't satisfy requirements

MOCKS INJECTOR / PROVIDER

AUTOMATIC INJECTION OF MOCKS FOR INTERFACES

CONFIGURATION

```
struct mocks_provider : di::config {
  struct mock_provider {
    template <class T, class TInit, class TMemory, class... TArgs>
    std::enable_if_t<!std::is_polymorphic<T>::value, T*> get(
    get(const TInit&, const TMemory&, TArgs&&... args) {
      return new T{std::forward<TArgs>(args)...};
    }

    template <class T, class TInit, class TMemory, class... TArgs>
    std::enable_if_t<std::is_polymorphic<T>::value, T*> get(
      return &get_mock<T>();
    }
  };

public:
  static auto provider(...) noexcept { return mock_provider{}; }
};
```

GET MOCK

https://github.com/eranpeer/FakeIt
https://github.com/dascandy/hippomocks

EXAMPLE

APP

```cpp
class iview {
public:
  virtual ~iview() noexcept = default;
  virtual void update() =0;
};
struct model { std::vector<int> board; };
class controller {
public:
  controller(model&, iview&) {}
};
struct user {};

class app {
public:
  app(controller&, user&) {}
};
```

INJECTOR

```
auto injector = di::make_injector<mocks_provider>(
  di::bind<int[]>.to({1, 2, 3, 4, 5, 6})
  // we don't have to bind iview!
);

injector.create<app>();
```

FAKE IT

```
auto&& v = get_mock<iview>();
When(Method(v, update)).AlwaysDo([]{...});
```

---

https://github.com/modern-cpp-examples/match3

MORE EXTENSIONS

Constructor Bindings | Contextual Bindings

XML Injection | Assisted Injection

Concepts | Lazy | Named Parameters

Types Dumper | UML Dumper | Serialize

Mocks Provider

Scoped Scope | Session Scope | Shared Scope

PERFORMANCE

ENVIRONMENT

2.3 GHZ INTEL CORE I7 / 16 GB 1600 MHZ DDR3

RUN-TIME

BIND TYPE TO VALUE

BINDINGS

```
#include <boost/di.hpp>

namespace di = boost::di;

auto test() {
  auto injector = di::make_injector(
    di::bind<int>.to(42)
  );

  return injector.create<int>();
}
```

ASM X86-64

```
mov $0x2a,%eax
retq
```

SAME AS

```
return 42;
```

HOW?

NO RUN-TIME BRANCHES

Everything is known at compile-time and may be optimized

CREATE (ASM X86-64)

```
injector.create<int>();  ---------------\
  create_successful_impl(type<int>());  |      mov $0x2a,%eax
    scope.create<int>(provider);        |--->  retq
      provider.get<int>();              |
        return 42; ---------------------/
```

BIND INTERFACE TO IMPLEMENTATION

BINDINGS

```cpp
#include <boost/di.hpp>

namespace di = boost::di;
auto test() {
  auto injector = di::make_injector(
    di::bind<interface>.to<implementation>()
  );

  return injector.create<std::unique_ptr<interface>>();
}
```

ASM X86-64

```
push    %rbx
mov     %rdi,%rbx
mov     $0x8,%edi
callq   0x4009f0 <_Znwm@plt>
movq    $0x400e78,(%rax)
mov     %rax,(%rbx)
mov     %rbx,%rax
pop     %rbx
retq
```

SAME AS

```
return std::make_unique<implementation>();
```

COMPILE-TIME

INCLUDE DI HEADER

DI.HPP

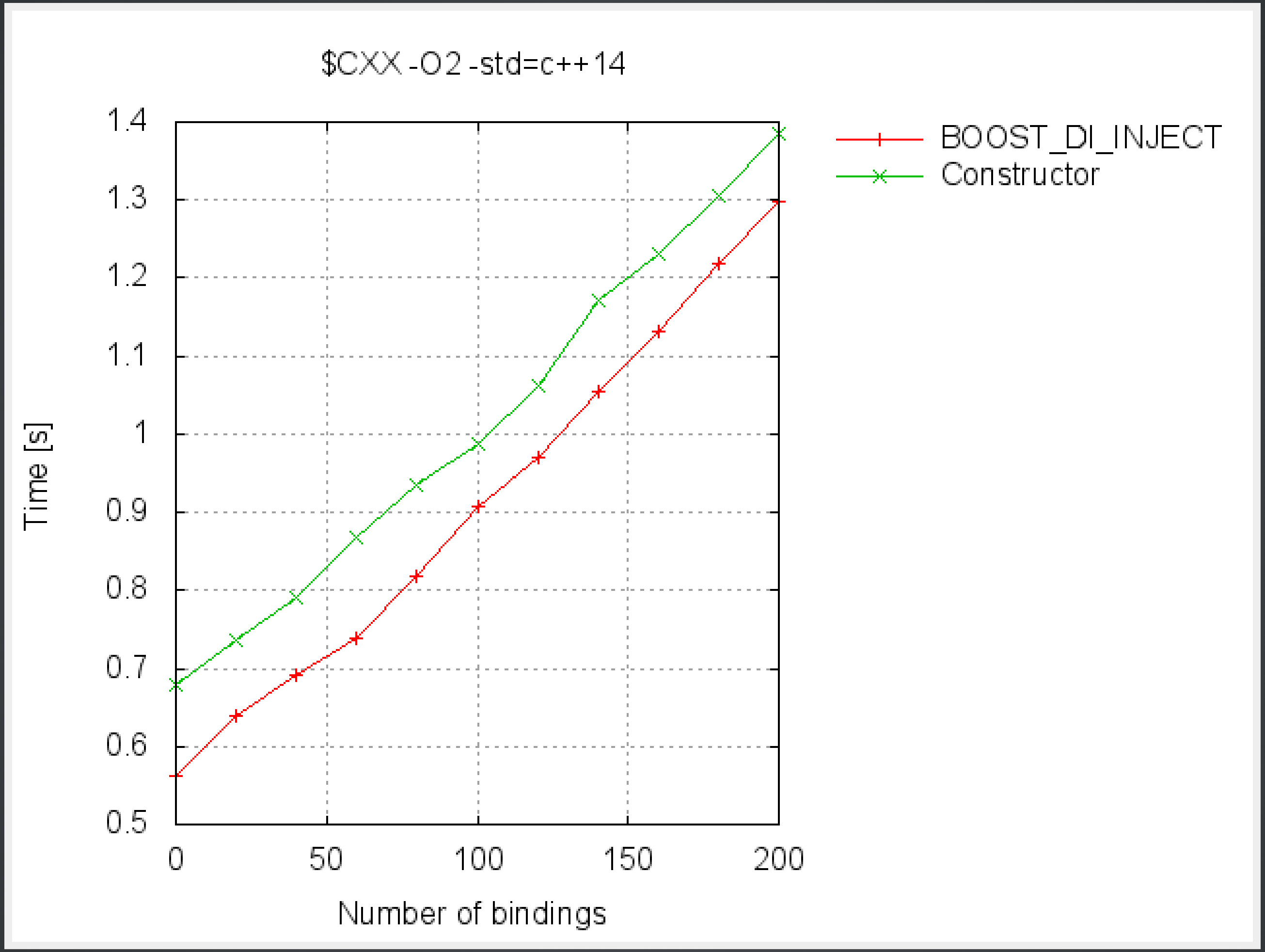
```
#include <boost/di.hpp>

int main() {}
```

```
$CXX -std=c++14 di.cpp # 0.050s
```

Neither STL nor Boost is required

CONSTRUCTION BENCHMARKS

CREATE BENCHMARK



```
4248897537 instances created
132 different types
10 modules
```

HOW QUICK COMPILATION TIMES WERE ACHIEVED?

ALWAYS MEASURE!

GUIDELINES

LIMIT TEMPLATE INSTANTIATIONS

AVOID 'NAIVE' RECURSIVE TEMPLATE ALGORITHMS

DO CHECKS ONCE AND AS EARLY AS POSSIBLE

AVOID LONG TYPE NAMES (VARIADIC TEMPLATES)

TAKE ADVANTAGE OF COMPILER BUILT-INS (VIA STL) AND COMMON TRICKS TO GAIN PERFORMANCE

```
template<bool...> struct bool_seq;

template<class... Ts>
using and_ = std::is_same<
    bool_seq<Ts::value...>,
    bool_seq<(Ts::value, true)...>
>;
```

```
__make_index_seq is O(1) since Clang 3.9
```

IMPLEMENTATION DETAILS

RESOLVE

INJECTOR DEPENDENCIES

```
template<class T>
struct dependency_concept { };

template<class I, class Impl>
struct dependency
  : pair<dependency_concept<I>, dependency<I, Impl>> { };

template<class... Ts>
struct injector : Ts...  { };

template<class... Ts>
auto make_injector(Ts...) {
  return injector<Ts...>{};
}
```

BINDER / RESOLVER

```
struct binder {
  template <class TDefault, class>
  static TDefault resolve_impl(...) noexcept { return {}; }

  template <class, class TConcept, class TDependency>
  static decltype(auto)
  resolve_impl(pair<TConcept, TDependency>* dep) noexcept {
    return static_cast<TDependency&>(*dep);
  }

  template <class T, class TDefault, class TDeps>
  static decltype(auto) resolve(TDeps* deps) noexcept {
    using dependency = dependency_concept<std::decay_t<T>>;
    return resolve_impl<TDefault, dependency>(deps);
  }
};
```

EXAMPLE

RESOLVE

```
auto injector = make_injector(dependency<i1, impl1>{});

struct default_dependency{};

static_assert(std::is_same<dependency<i1, impl1>,
  std::decay_t<decltype(
    binder{}.resolve<i1, default_dependency>(&injector))
  >>{}
);

static_assert(std::is_same<default_dependency,
  std::decay_t<decltype(
    binder{}.resolve<i2, default_dependency>(&injector)
  )>>{}
);
```

http://melpon.org/wandbox/permlink/yMIaCIIDtjBXmV0E

RESOLVE - BENCHMARK

Number of dependencies	Resolve dependencies (all)
1	0.077s
16	0.079s
32	0.082s
64	0.083s
128	0.089s

CREATABLE CONCEPT

IDEA

```
                    creatable<T>
                         |
          /--------------+-----------------\
        Yes                                No
         |                                 |
create_successful_impl                 create_impl
// No error checking                // Error checking
```

CREATABLE CONCEPT

```
template <class T> requires creatable<T>()
constexpr T create() const {
  // no checks for errors!
  return create_successful_impl(type<T>{});
}

template <class T> requires !creatable<T>()
[[deprecated("creatable constraint not satisfied")]]
constexpr T create() const {
  // checks for errors to report it
  return create_impl(type<T>{});
}
```

'TYPE-NAME' ERASURE

LONG TYPE NAMES MAY INCREASE YOUR COMPILATION TIMES BY A HUGE FACTOR!

LONG TYPE-NAME DUE TO VARIADIC TEMPLATES

```
template<class... Ts> auto make_injector(Ts... args) {
  return injector<Ts...>(args...); // may produce a long type name
}

auto injector = make_injector(...);
injector.create<T>(); // compiles slowly due to
                      // long type names comparisons
```

SOLUTION - INHERITANCE

Hide the long type name

TYPE-NAME ERASURE

```
auto injector = di::make_injector(...);

struct erase_long_type_name : decltype(injector) {
  using decltype(injector)::injector;
};

erase_long_type_name{injector}.create<T>(); // quick to compile
                                            // type name =>
                                            // erase_long_type_name
```

PROBLEM - IT'S NOT FLEXIBLE

It has to be done from 'non long type name' context -> user context

SOLUTION - LAMBDA EXPRESSION

TYPE-NAME ERASURE

```
static auto make_injector_impl = [](auto injector) {
  using injector_t = decltype(injector);

  struct i : injector_t {
    explicit i(injector_t&& other)
      : injector_t(std::move(other)) { }
  };

  return i{std::move(injector)};
};
```

'TYPE-NAME' ERASURE - BENCHMARK

http://melpon.org/wandbox/permlink/aot9ePGgtKtVKVKP

Solution	Number of bindings	Time
Long Type-Name	256	5.321s
Type-Name erasure / inheritance	256	2.521s
Type-Name erasure / lambda expression	256	3.278s

MORE BENCHMARKS

http://boost-experimental.github.io/di/overview

DI VS ISO C++

MISSING FEATURES

STATIC REFLECTION

DEDUCTION OF CONSTRUCTOR PARAMETERS

```
class example {
public:
  example(int, double);
};

static_assert(std::is_same<
  std::tuple<int, double>
, std::function_traits_t<decltype(&example::example)>::args
>{});
```

USER DEFINED ATTRIBUTES

SELECT CONSTRUCTOR

```
class example {
public:
  [[inject]]
  example(double, int); // pick me!

  example(int, double);
  example(int, double, float);
};
```

NAMED PARAMETERS

```
class example {
public:
  example([[named("int_a")]] int a, [[named("int_b")]] int b) {
    assert(42 == a);
    assert(87 == b);
  }
};
```

BINDINGS

```
auto injector = di::make_injector(
  di::bind<int>.named("int_a"_s).to(42)
, di::bind<int>.named("int_b"_s).to(87)
);
```

WHERE

COMPILE-TIME NAME

```
"int_a"_s
"int_b"_s
```

STRING-LITERAL-OPERATOR-TEMPLATE

```
template <class T, T... Ts>
constexpr auto operator""_s();
```

RETRIEVE A CONCEPT TYPE

DUMMY CONCEPT

```
template <typename T>
concept bool Dummy() {
  return requires(T t) {
    { t.dummy() };
  };
}

struct DummyImpl {
  void dummy() {}
};
```

```
struct app {
  app(Dummy); // => template<class T> app(T) requires Dummy<T>()
};            // T vs Dummy?
```

BIND CONCEPT TO A TYPE

```
di::bind<Dummy>.to<DummyImpl>() // not possible
```

QUESTIONS?

- Documentation
  - http://boost-experimental.github.io/di
- Source Code
  - https://github.com/boost-experimental/di
- Try it online
  - http://boost-experimental.github.io/di/try_it

THANK YOU