High performance web programming with C++14

Matthieu Garrigues, ENSTA-ParisTech

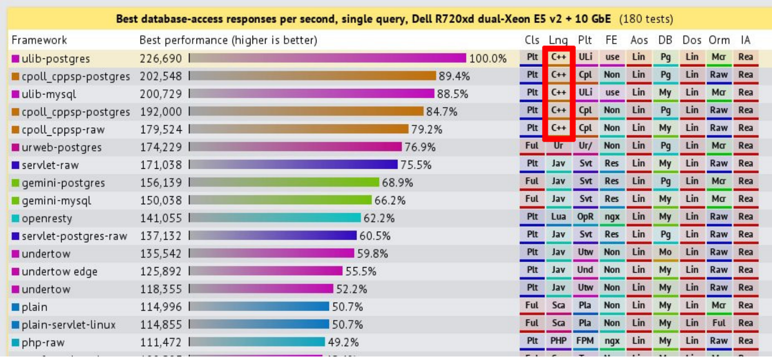
May 13, 2015

Outline

Outline

Motivations

C++ is one of the most efficient language for web programming: `techempower.com/benchmarks`



Best database-access responses per second, single query, Dell R720xd dual-Xeon E5 v2 + 10 GbE (180 tests)

Framework	Best performance (higher is better)		Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA
ulib-postgres	226,690	100.0%	Plt	C++	ULi	use	Lin	Pg	Lin	Mcr	Rea
cpoll_cppsp-postgres	202,548	89.4%	Plt	C++	Cpl	Non	Lin	Pg	Lin	Raw	Rea
ulib-mysql	200,729	88.5%	Plt	C++	ULi	use	Lin	My	Lin	Mcr	Rea
cpoll_cppsp-postgres	192,000	84.7%	Plt	C++	Cpl	Non	Lin	Pg	Lin	Raw	Rea
cpoll_cppsp-raw	179,524	79.2%	Plt	C++	Cpl	Non	Lin	My	Lin	Raw	Rea
urweb-postgres	174,229	76.9%	Ful	Ur	Ur/	Non	Lin	Pg	Lin	Mcr	Rea
servlet-raw	171,038	75.5%	Plt	Jav	Svt	Res	Lin	My	Lin	Raw	Rea
gemini-postgres	156,139	68.9%	Ful	Jav	Svt	Res	Lin	Pg	Lin	Mcr	Rea
gemini-mysql	150,038	66.2%	Ful	Jav	Svt	Res	Lin	My	Lin	Mcr	Rea
openresty	141,055	62.2%	Plt	Lua	OpR	ngx	Lin	My	Lin	Raw	Rea
servlet-postgres-raw	137,132	60.5%	Plt	Jav	Svt	Res	Lin	Pg	Lin	Raw	Rea
undertow	135,542	59.8%	Plt	Jav	Utw	Non	Lin	Mo	Lin	Raw	Rea
undertow edge	125,892	55.5%	Plt	Jav	Und	Non	Lin	My	Lin	Raw	Rea
undertow	118,355	52.2%	Plt	Jav	Utw	Non	Lin	My	Lin	Raw	Rea
plain	114,996	50.7%	Ful	Sca	Pla	Non	Lin	My	Lin	Mcr	Rea
plain-servlet-linux	114,855	50.7%	Ful	Sca	Pla	Non	Lin	My	Lin	Ful	Rea
php-raw	111,472	49.2%	Plt	PHP	FPM	ngx	Lin	My	Lin	Raw	Rea

Motivations

But, it is not really famous for its productivity and ease of use.

Motivations

So, let's leverage the new C++14 to ease the writting of web services without impacting its speed of execution...

Outline

C++11/14

The C++11/14 greatly improves C++, but C/C++ web frameworks were created with C++98:

- TreeFrog
- cppnetlib
- Wt
- CppCMS
- lwan
- h2o
- Facebook Proxygen
- ....

C++11/14

Let's take a look at some of the C++11/14 new features.

C++11/14: auto

Automatic type inference

```
auto i = 42;

for (auto it = v.begin(); it != v.end(); it++)
{ ... }

template <typename A, typename B>
auto fun(A a, B b) { return a + b; }
```

C++11/14: decltype

Automatic type computation

```
using a_type =
   decltype(/a complex expression.
            You do not want to manually compute its
                type./)
```

C++11/14: lambda functions

Lambda functions

```
std::vector<int> V = { 3, 6, 2, 5, 6, 7, 5};

std::sort(V.begin(), V.end(),
          [] (int a, int b) { return a > b; });
```

C++11/14: What's new?

Generic lambda functions

```
auto print = [] (auto e)
{
  cout << e << endl;
};

print(1); // int
print(2.3f); // float
print("test"); // const char*
```

C++11/14: What's new?

Variadic templates

```
void variadic_printf() {}

template <typename A, typename... T>
void variadic_printf(const A& a, T&&... tail)
{
  std::cout << a;
  variadic_printf(std::forward<T>(tail)...);
}
```

C++11/14: What's new?

The constexpr keyword

```
constexpr int
compile_time_add(int a, int b)
{
  return a + b;
}
```

C++11/14: What's new?

They are great features. But how do they help Web Programming?

Outline

Our problem

A kind of static introspection exists in C++.
But, it does not help to build:

- Automatic serialization / deserialization
- Object relational mapping

$\Rightarrow$ Let's improve C++ introspection.

Definition of a symbol

A symbol is a meta object member carrying its own static introspection data.
Let dig into the symbol _car:

```
struct _car_t
{
  // symbol to string.
  const char* name() { return "car"; }

  template <typename T>
  struct variable_type // Meta variable
  {
    T car; // car member.
    using symbol_type = _car_t;
    auto symbol() const { return _car_t(); }
  };
}

_car_t _car; // Symbol definition.
```

Symbol definition

By convention symbols start with _. They are included in the namespace s to avoid name conflicts.

A macro function helps the definition of symbols:

```
iod_define_symbol(car);  // defines s::_car
iod_define_symbol(name); // defines s::_name
```

Using symbols

Using the symbol car:

```
auto x = _car_t::variable_type<string>();
x.car = "BMW";
// x.value() == "BMW"
// x.symbol() returns _car_t();
// x.symbol().name() returns "car";
```

Only one member?

Just one member per object is quite a limitation, so...

Statically introspectable objects

Let's stack them together into IOD's statically introspectable objects (SIO).

```
template <typename... Members>
struct sio : public Members...
{
  sio(Members... s) : Members(s)... {}
};

using person_type =
   sio<_id_t::variable_type<int>,
       _name_t::variable_type<string>>;
```

IOD relies on inheritance to stack the members id and name together.

Statically introspectable objects

The `D` helper is our friend.

```
auto john = D(_id = 42,
              _name = "John");

// john.id == 42;
// john.name == "John";
```

Inside SIOs

Behind the scene, D puts the introspection data in the SIO type:

```
decltype(john)
```

==

```
iod::sio<s::_id_t   ::variable_type<int>,
         s::_name_t ::variable_type<string>>
```

Foreach

Then, `iod::foreach` can easily iterate on statically introspectable objects.
Let's write a generic serializer:

```
foreach(any_sio_object) | [] (auto& m)
{
  std::cout << m.symbol().name()
            << ": " << m.value() << std::endl;
};
```

=> Unrolled at compile time, no runtime cost.

Foreach

`iod::foreach` also handles multiple arguments, the creation of new objects...

```
auto sum =
  foreach(o1, o2) | [] (auto& m1, auto& m2)
  {
    return m1.symbol() = m1.value() + m2.value();
  };
```

Note: o1 and o2 must have the same number of members.

Foreach

... and tuples

```
auto sum =
  foreach(tuple1, tuple2) | [] (auto& e1, auto& e2)
  {
    return e1 + e2;
  };
```

Note: tuple1 and tuple2 must have the same number of elements.

Static instropection: What for?

On top of static introspection (and other utilities), the IOD library implements:

- JSON serialization / deserialization
- Dependency injection

Outline

The Silicon Web Framework: Goal

The Silicon Web Framework leverages **static introspection** to **ease** the writting of web services, without impacting the **performances**.

The Silicon Web Framework: Hello world

Let's build a simple hello world api.

The Silicon Web Framework: Hello world

To serve this simple procedure via http:

```
[] { return "hello world"; }
```

The Silicon Web Framework: Hello world

Wrap it in an API and map the function to the route /hello:

```
make_api(_hello = [] { return "hello world"; });
```

Note the use of IOD statically introspectable objects to model the API.

The Silicon Web Framework: Hello world

Let's launch the microhttpd HTTP backend to serve our API. Because the API is actually a SIO, the backend can bind the route `/hello` to our lambda function.

```
mhd_json_serve(make_api(_hello = [] {
                    return "hello world";
               }), 9999);
```

That's it.

```
curl "http://127.0.0.1:9999/hello"
hello world
```

Up to 285000 requests/seconds on a 4 cores Intel I5 3GHz: Exactly what you get with a plain C microhttpd hello world server.

The Silicon Web Framework: Procedure Arguments?

Procedures can take arguments:

```
auto api = make_api(
  _hello(_name = string()) = [] (auto params) {
      return "hello " + params.name; }
);
```

The backend is responsible for deserialization and validation of the procedure arguments.

The Silicon Web Framework: Returning objects

Procedures can also return statically introspectable objects:

```
auto api = make_api(
  _hello(_name = string()) = [] (auto params) {
      return D(_message = "Hello" + params.name); }
);
```

The backend is responsible for serialization of the procedure return values.

Middlewares And Dependency injection

We need to provide access to middlewares:

- Databases
- Sessions
- Logging
- ...

Middlewares And Dependency injection

However, not all procedures need an access to all middlewares.

Middlewares And Dependency injection

We want to require access to the middlewares just by declaring them as argument:

```
auto api = make_api(
  _a_procedure = [] (sqlite_connection& c,
                     logger& l) {
    // ...
  }
);
```

The framework introspects the function signature to inject the matching middlewares as arguments.

Middlewares And Dependency injection

Most middlewares cannot be created from nothing. Some need factories.

The `bind_factories` method attaches factories to a given API:

```
auto api = make_api(
  _procedure1 = [] (sqlite_connection& c) {},
  _procedure2 = [] (my_logger& l) {},
  _procedure3 = [] (my_logger& l, sqlite_connection& c)
      {}
  ).bind_factories(
      sqlite_connection_factory("blog.sqlite"),
      my_logger_factory("/tmp/server.log")
  );
```

IOD's dependency injection takes care of binding the right factory to the right procedure arguments.

Anatomy of middleware

A middleware factory is a plain C++ class with one `instantiate` method: the dependency injection entry point.

Let's have a look at `session_factory::instantiate`:

```
session instantiate(cookie& ck, db_connection& con)
{
  return session(ck, con, this->sql_session_table);
}
```

- `session` is the middleware type.
- Its instantiation depends on two middlewares: `cookie` and `db_connection`.

Inter Middleware Dependencies

`cookie` and `db_connection` also have factories, and their instantiation may depend on other middlewares.

⇒ This leads to a dependency tree, resolved by IOD's dependency injection at compile time.

Inter Middleware Dependencies

In other words, if a procedure requires a session object:

```
auto api = make_api(_procedure1 = [] (session& s) {});
```

Inter Middleware Dependencies

The framework generates a code similar to this:

```
cookie ck = cookie_factory.instantiate();
db_connection con= db_connection_factory.instantiate();
session s = session_factory.instantiate(ck, con);

api.procedure1(s);
```

SQL Middlewares

Silicon SQL middlewares provide a basic interface with SQL databases.
Sqlite and **MySQL** are already available.

```
[] (sqlite_connection& c) {
  int i;
  c("Select 1 + 2") >> i;
}
```

SQL Middlewares: Iterations on Result Sets

Straightforward iteration on a result set:

```
c("SELECT name, age from users")() |
  [] (std::string& name, int& age)
  {
    std::cout << name << " " << age << std::endl;
  };
```

Again, introspection on the lambda function arguments helps to write zero cost abstractions.

SQL Middlewares: Prepared statements

For better performance, prepared statements are created, cached, and reused whenever a request is triggered more than once.

```
// First call, prepare the SQL statement.
c("INSERT into users(name, age) VALUES (?, ?)")
 ("John", 12);

// Second call, reuse the cached statement.
c("INSERT into users(name, age) VALUES (?, ?)")
 ("Bob", 14);
```

SQL ORM Middlewares

It is pretty easy to generate SQL requests from statically introspectable objects.
Let's declare our statically introspectable `User` data type:

```
typedef
decltype(D(_id                  = int(),
           _login               = string(),
           _password            = string()))
  User;
```

SQL ORMs

And give the ORM hints for the SQL generation:

```
typedef
decltype(D(_id(_auto_increment,
               _primary_key) = int(),
           _login          = string(),
           _password       = string()))
  User;
```

SQL ORM Middlewares: Example

```
// The middleware type.
typedef sqlite_orm<User> user_orm;
typedef sqlite_orm_factory<User> user_orm_factory;

// An API making use of the ORM.
auto api = make_api(
  _test_orm(_id = int()) = [] (auto p, user_orm& users)
  {
    User u = users.find_by_id(p.id);
    u.login = "Rob";
    users.update(u);
  }
// The factories.
).bind_factories(sqlite_connection_factory("db.sql"),
                 user_orm_factory("user_table"));
```

SQL CReate Update Delete

`sql_crud` generates Create Update and Delete routes for a given ORM type:

```
auto api = make_api(_user = sql_crud<user_orm>());
mhd_json_serve(api, 9999);
```

Creates the following routes:

```
/user/create
/user/update
/user/destroy
```

And saves you tens of lines of code.

SQL CReate Update Delete

The parameters of `sql_crud` allow to handle:

- validation of objects
- user authentication
- pre/post processing.

SQL CReate Update Delete

This parameterization relies on lambda functions and dependency injection (DI). Let's write an example with the `_write_access` option:

```
sql_crud<user_orm>(
  _write_access = [] (user& u, session& s) {
      return u.id == s.user_id;
  }
)
```

Thanks to dependency propagation, the middlewares (`session` in this example) and the current `user` object are accessible from the callback.

Silicon Backends

We have APIs and middlewares, let's plug everything into the network.

Silicon Backends

Silicon backends leverage the introspection on Silicon APIs to serve them via:

- Differents protocols: HTTP/1, HTTP/2, Websockets...
- Differents message formats: JSON, XML, ...
- Differents protocol implementations: microhttpd, h2o, ...

As of today, April 2015, Silicon includes microhttpd/json and websocketpp/json.

Limitations of the framework

Slow compilation

- **2s** for the hello world one liner
- **35s** for a complex 110 lines API

GCC error messages

- Static introspection can generate very, very long types
- GCC error messages get hard to digest
- Much better with Clang shortening long types

Future works

- Lower compilation time
- More database middlewares (PostgreSQL, Redis, ...)
- More backends (HTTP/2, ...)
- Suggestions?

$\Rightarrow$ Contributions are welcome

Outline

Conclusion

The new C++ features enabled us to build zero cost abstractions for building web services:

- Simple to write
- Without impacting running time
- Where most bugs are reported by the compiler at compile time
- With a tiny framework of less than 10000 C++ lines.
- Open source (MIT): `github.com/matt-42/silicon`

Question?

```
_ask(_question) = [] (auto p, my_smart_middleware& m)
{
  return m.answer(p.question);
}
```

Or visit http://siliconframework.org