JSON meets metaprogramming

basel[devs].find("daminetreg").talk()

What is JSON ?

- It’s a less complete YAML =p

```
{
  "interests": ["sport articles"],
  "money_spent":50,
  "name":"Mrs. Fraulein"
}
```

What is metaprogramming ?

- The *meta* prefix means *about*.
  - A program reasoning about a program.
  - Input data is code, output data is code.
  - Can C++ do that ?

Let them meet !

The nice refreshing JSON

and

the obscure metaprogramming techniques meet.

If they don’t meet :

```
struct person {
    std::string name;
    std::string address;
    int age;
};

person from_json(const json& j) {
    person p;
    p.name = j.at("name").get<std::string>();
    p.address = j.at("address").get<std::string>();
    p.age = j.at("age").get<int>();
    return p;
}

json to_json(person& p) {
    json j;
    j["name"] = p.name;
    j["address"] = p.address;
    j["age"] = p.age;
     return j;
}
```

Why repeating ourselves ?

- Don’t we want to party with friends, instead of fixing bugs in `to_json` & `from_json` ?

- There are too much JSON libraries :
  - A friend of mine once made one, and named it KFATHER to avoid name clash.
  - But still we keep repeating ourselves.

- A metaprogram could code `from_json` and `to_json` for us.

How do we model JSON in C++ ?

```
struct cook {
  int michelin_stars;
};

struct dev {
  std::vector<std::string> programming_languages;
};

struct secretary {
  std::vector<std::string> spoken_languages;
};

struct housekeeper {
  bool can_do_electricity;
  bool can_do_plumbing;
  bool can_garden;
  bool can_wash;
};

using job_t = boost::variant<cook, dev, secretary, housekeeper>;
```

```
struct person {
  std::string name;
  std::string address;
  int age;
  job_t job;
};
```

How do we model JSON in C++ ?

```
struct cook {
  int michelin_stars;
};

struct dev {
  std::vector<std::string> programming_languages;
};

struct secretary {
  std::vector<std::string> spoken_languages;
};

struct housekeeper {
  bool can_do_electricity;
  bool can_do_plumbing;
  bool can_garden;
  bool can_wash;
};

using job_t = boost::variant<cook, dev, secretary, housekeeper>;
```

["c++", "swift", "js"]

```
struct person {
  std::string name;
  std::string address;
  int age;
  job_t job;
};
```

{ "programming_languages" : [...] }

from_json

- to_json is too easy.
- Writing from_json so that it automagically works for any type.
- As Efficient as written by hand.

from_json the hard way

```
inline person from_json(const json& j) {
  person p;
  p.name = j.at("name").get<std::string>();
  p.address = j.at("address").get<std::string>();
  p.age = j.at("age").get<int>();

  if (j["job"]["struct"].get<std::string>() == "bs::dev") {
    auto job = dev{};

    for (auto e : j["job"]["programming_languages"]) {
      job.programming_languages.push_back(e.get<std::string>());
    }
    p.job = job;
  } else if (j["job"]["struct"].get<std::string>() == "bs::secretary") {
    auto job = secretary{};

    for (auto e : j["job"]["spoken_languages"]) {
      job.spoken_languages.push_back(e.get<std::string>());
    }
    p.job = job;
  } else if (j["job"]["struct"].get<std::string>() == "bs::cook") {
    auto job = cook{};
    job.michelin_stars = j["job"]["michelin_stars"].get<int>();
    p.job = job;
  }

  return p;
}
```

from_json the hard way

```
inline person from_json(const json& j) {
  person p;
  p.name = j.at("name").get<std::string>();
  p.address = j.at("address").get<std::string>();
  p.age = j.at("age").get<int>();

  if (j["job"]["struct"].get<std::string>() == "bs::dev") {
    auto job = dev{};

    for (auto e : j["job"]["programming_languages"]) {
      job.programming_languages.push_back(e.get<std::string>());
    }
    p.job = job;
  } else if (j["job"]["struct"].get<std::string>() == "bs::secretary") {
    auto job = secretary{};

    for (auto e : j["job"]["spoken_languages"]) {
      job.spoken_languages.push_back(e.get<std::string>());
    }
    p.job = job;
  } else if (j["job"]["struct"].get<std::string>() == "bs::cook") {
    auto job = cook{};
    job.michelin_stars = j["job"]["michelin_stars"].get<int>();
    p.job = job;
  }

  return p;
}
```

You said JSON is simple ?

from_json the easy way

```
#include <pre/json/from_json.hpp>
#include <pre/json/to_json.hpp>

BOOST_FUSION_ADAPT_STRUCT(bs::cook, michelin_stars)
BOOST_FUSION_ADAPT_STRUCT(bs::dev, programming_languages)
BOOST_FUSION_ADAPT_STRUCT(bs::secretary, spoken_languages)

BOOST_FUSION_ADAPT_STRUCT(bs::housekeeper,
  can_do_electricity, can_do_plumbing, can_garden, can_wash)

BOOST_FUSION_ADAPT_STRUCT(bs::person, name, address, age, job)

pre::json::from_json<person>(some_input);
```

Great but how does it works ?

- BOOST_FUSION_ADAPT_STRUCT is a preprocessor metaprogram generating queriable tree of types.

- pre::json::from_json<T> is another metaprogram generating a from_json function out of a tree of types.

- Let’s dig in !

Could it be better ?

- `constexpr if` would allow simplifying the code

- We could get rid of the BOOST_FUSION_ADAPT_STRUCT with the `reflexpr operator`

Could it be better ?

- `constexpr if` would allow simplifying the code
- We could get rid of the BOOST_FUSION_ADAPT_STRUCT with the `reflexpr operator`
- If you mind contributing !
  - https://daminetreg.github.io/lib-cpp-pre