Named template parameters

Warsaw C++ User Group

Tomasz Żołnowski

12 listopada 2013

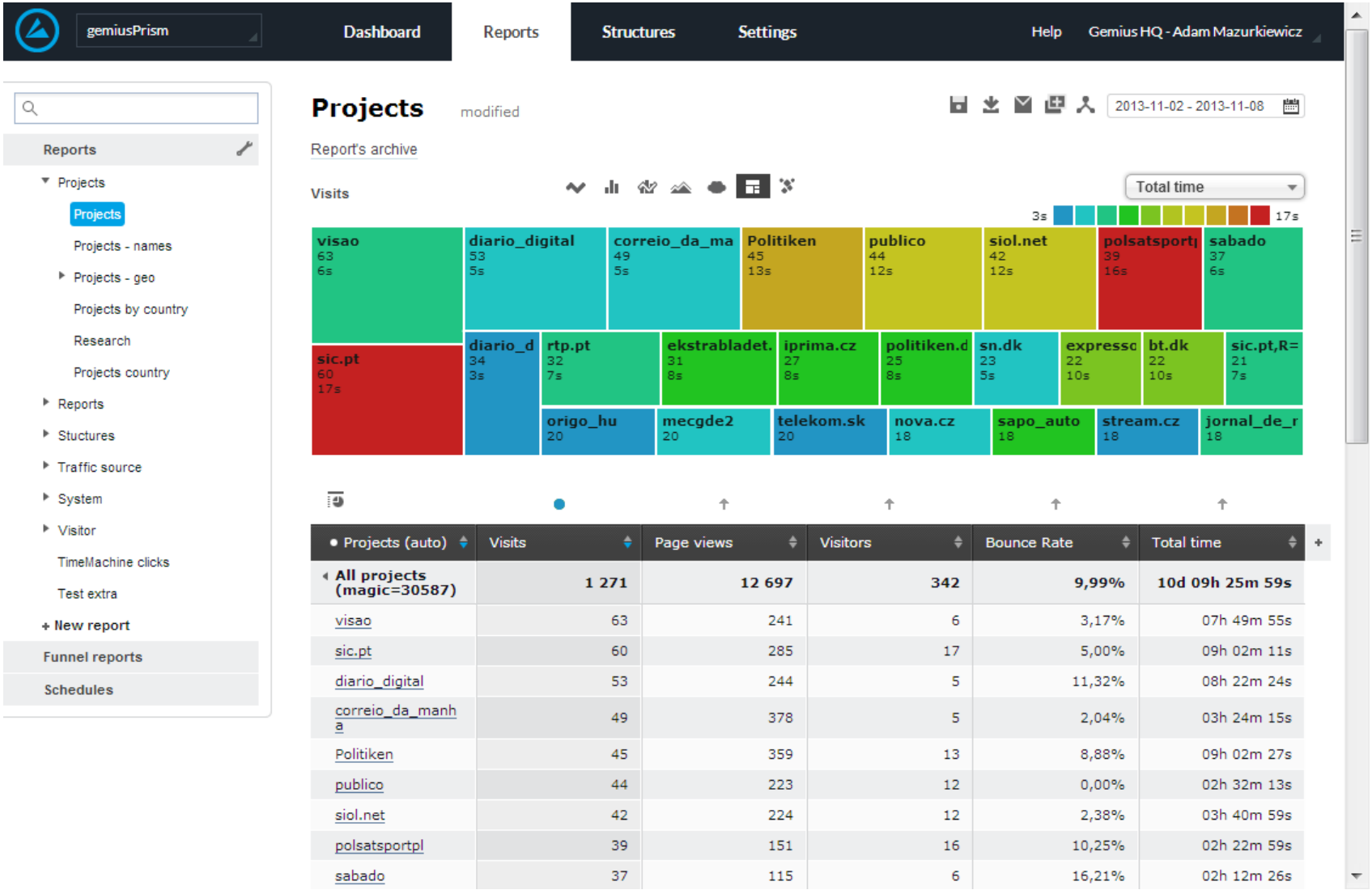
Gemius

Krótko o projektach w firmie.

Gemius

Produkty:

- gemiusAudience
- gemiusPrism
- gemiusTraffic
- gemiusDirectEffect
- gemiusHeatMap
- gemiusStream
- gemiusShopMonitor



Projects (auto)	Visits	Page views	Visitors	Bounce Rate	Total time
All projects (magic=30587)	**1 271**	**12 697**	**342**	**9,99%**	**10d 09h 25m 59s**
visao	63	241	6	3,17%	07h 49m 55s
sic.pt	60	285	17	5,00%	09h 02m 11s
diario_digital	53	244	5	11,32%	08h 22m 24s
correio_da_manha	49	378	5	2,04%	03h 24m 15s
Politiken	45	359	13	8,88%	09h 02m 27s
publico	44	223	12	0,00%	02h 32m 13s
siol.net	42	224	12	2,38%	03h 40m 59s
polsatsportpl	39	151	16	10,25%	02h 22m 59s
sabado	37	115	6	16,21%	02h 12m 26s

Gemius

Projekty niskopoziomowe:

- MooseFS
  rozproszony system plików
- Taskell
  system do rozpraszania zadań i kontroli zasobów
- VSender
  główna baza danych

VSender

- zbierane są odsłony
- ...dużo odsłon, 12 miliardów dziennie
- pliki dobowe powyżej 200G
- ...po kompresji 1:40
- jedna odsłona zajmuje kilkanaście bajtów
- dane są sortowane, składowane i serwowane z małym opóźnieniem
- system korzysta z MooseFS i Taskell

CommonLib

- logowanie zdarzeń
- obsługa puli wątków
- system wyjątków
- interfejs MySQL
- interfejs SQLite
- serwis geolokalizacji
- serwis stref czasowych
- narzędzia systemowe
- RTL - format plików naszych baz danych

RTL

- paradygmat const database
- pliki systemu VSender są w tym formacie
- wykorzystywana biblioteka **boost**
- zawiera algorytmy kompresji, serializacji,
- format pliku:
  - dane podzielone na bloki
  - rekordy zgodnie z zadanym kluczem
  - indeks po kluczu głównym
- CreateClass - zestaw makr ułatwiający definicję rekordu

Definicja rekordu:

```
CREATE_CLASS(
        MyRecord
    ,
        ((ID)   (uint64_t)      (DEFAULT))
        ((TS)   (int32_t)       (DEFAULT))
        ((Name) (std::string)   (DEFAULT))
    ,
        PRINTABLE
        DEFAULTCONSTRUCTIBLE
    )
```

Konfiguracja bazy przez parametry szablonu:

```
template <
    typename RecordType,
    typename KeyFunctor,
    template <class> class Codec,
    typename EquivJoinPolicy,
    typename Defaults,
    typename IndexType,
    typename BlockFlushPolicy,
    typename SorterBufferPolicy,
    bool Stable
>
class DatabaseConfig;
```

Analogiczny przykład z języka Python:

```
subprocess.check_call(
    args, *,
    stdin=None,
    stdout=None,
    stderr=None,
    shell=False)
```

Wiele opcji, a można:

```
subprocess.check_call(
    ['seq', '1', '5'],
    stdout=open('output.txt', 'w'))
```

Jak coś takiego zrobić ogólnie dla funkcji w C++?

- przeciążanie `operator=`
- przeciążanie `operator,`
- `foo((_a=5, _b=10));`
- `foo(const Opts& = Opts());`
- trudniejsze ze względu na referencje
- można użyć **boost.parameter**

W przypadku szablonów:

- zbliżone do programowania funkcyjnego
- i dzięki temu łatwiejsze

Istnieje rozwiązanie książkowe:

C++ Templates: The Complete Guide
David Vandevoorde, Nicolai M. Josuttis

- oparte o dziedziczenie
- wirtualne dziedziczenie po klasie Defaults

Rozwiązanie ma pewne wady:

- nie można dziedziczyć (`public`) po tej samej klasie
- sporo nadmiarowych klas
- utrata kontroli przez mechanizm dziedziczenia
- u nas nie wszystkie cechy to `typename`/`typedef`
- część parametrów jest od siebie zależna

Zastosowane rozwiązanie nieksiążkowe:

- oparte o proste mechanizmy języka
- na podstawie listy typów
- szybko się kompiluje :)

- lista typów

```
template <typename Param, typename Tail>
struct ParamList { };

struct ParamEOList { };
```

- pusty parametr

```
struct NopParam { };
```

`DatabaseConfig`:

- `RecordType` i `KeyFunctor` - bez zmian
- kolejne `Param0`, `Param1`, itd. - jako zwykłe `typename`
- domyślnie `NopParam`
- nawlekamy na listę (w odwrotnej kolejności):

```
...
    ParamList<Param2,
        ParamList<Param1,
            ParamList<Param0, ParamEOList>
        >
    >
...
```

Kod można wygenerować, wykorzystamy **boost.preprocessor**:

```
#define CONFIG_PARAMS_COUNT 10

#define CONFIG_TEMPLATE_PARAMS ...

#define CONFIG_PARAMS_LIST ...
```

Generowanie parametrów szablonu:

```
#define CONFIG_MAKE_PARAM(Z, I, Data) \
    BOOST_PP_COMMA_IF(I) typename Param ## I = NopParam

#define CONFIG_TEMPLATE_PARAMS \
    BOOST_PP_REPEAT(CONFIG_PARAMS_COUNT, CONFIG_MAKE_PARAM, _)
```

Generowanie listy parametrów:

```
#define CONFIG_PARAMS_NODE_LHS(Z, I, Data) \
    ParamList<BOOST_PP_CAT(Param, \
            BOOST_PP_SUB(BOOST_PP_SUB(CONFIG_PARAMS_COUNT, I), 1)),

#define CONFIG_PARAMS_NODE_RHS(Z, I, Data) >

#define CONFIG_PARAMS_LIST \
    BOOST_PP_REPEAT(CONFIG_PARAMS_COUNT, CONFIG_PARAMS_NODE_LHS, _) \
        ParamEOList \
    BOOST_PP_REPEAT(CONFIG_PARAMS_COUNT, CONFIG_PARAMS_NODE_RHS, _)
```

Każdy parametr ma swój sposób definiowania:

```
struct UseFeature1 { };

struct UseFeature2 { };

template <typename Other>
struct SetFeature { };
```

...i swój mechanizm wyciągania wartości:

```
template <typename Params>
struct ExtractFeature;
```

Niech udostępnia np.: `typedef feature_type`.

...i częściowe specjalizacje:

a) trafiamy na nasz `feature`, np:

```
template <typename Tail>
struct ExtractFeature<ParamList<UseFeature1, Tail> >
{
    typedef feature1 feature_type;
};
```

...i częściowe specjalizacje:

b) trafiamy na `feature` użytkownika:

```
template <typename Other, typename Tail>
struct ExtractFeature<ParamList<SetFeature<Other>, Tail> >
{
    typedef Other feature_type;
};
```

...i częściowe specjalizacje:

c) ustawienie dla innej cechy:

```
template <typename Param, typename Tail>
struct ExtractFeature<ParamList<Param, Tail> >
     : ExtractFeature<Tail>
{
};
```

...i częściowe specjalizacje:

d) ustawienie domyślne:

```
template <>
struct ExtractFeature<ParamEOList>
{
    typedef default_type feature_type;
};
```

Użycie w DatabaseConfig:

```
template <
    typename RecordType,
    typename KeyFunctor,
    CONFIG_TEMPLATE_PARAMS
>
class DatabaseConfig
{
    typedef CONFIG_PARAMS_LIST Params;

    typedef typename ExtractFeature<Params>::feature_type
                feature_type;

    /* ... */
```

Przykład użycia.

Wcześniejszą konfigurację zapisywaliśmy:

```
typedef DatabaseConfig<
            TestRecord, TestKeyFunctor,
        NetworkOrderCodec,
        JoinEquivalentWith<TestJoiner>,
        RecordDatabaseDefaults<3, 4096>,
        StandardFileIndex<TestKeyFunctor::result_type, NetworkOr
        StandardBlockFlushPolicy<TestRecord, TestKeyFunctor>,
        JoinEquivalentWith<TestJoiner>::record_sorter_buffer_pol
        true
    > TestConfig;
```

...a teraz tak:

```
typedef DatabaseConfig<
            TestRecord, TestKeyFunctor,
        SetJoinPolicy<JoinEquivalentWith<TestJoiner> >,
        SetStable
    > TestConfig;
```

Pytania?

Dziękuję za uwagę.