

It's time to get

boost
C++ LIBRARIES



М. Масягин ИУ9-52(Б)

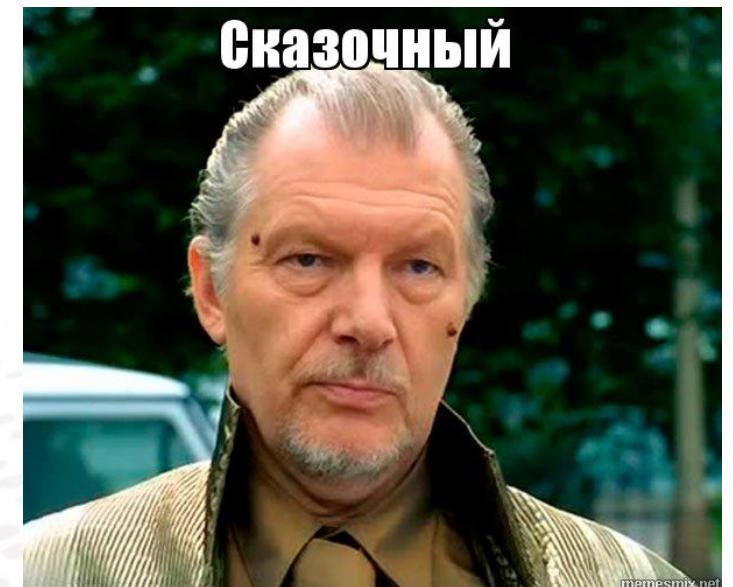
Что такое Boost и как он появился?

Однажды два C++ программиста, приняв пару стаканов вина, обсуждали разработку открытых библиотек, которые должны были бы содержать всё необходимое, не включенное в недавно вышедший Стандарт. Один из них упомянул, что Герб Саттер готовил пропозицию языка программирования Booze, который должен был быть лучше, чем Java. Смысл этой остроумной шутки в том, что java — сорт кофе, а booze — «бухло». Продолжением игры слов стало название «Boost» для набора открытых библиотек, куда на сегодняшний день вошли около сотни библиотек, а некоторые из них даже были запилены в нынешний или будущий Стандарт... (с) Lurk



На сегодняшний день Boost – это:

- Овердофига часов компиляции и ужасающий размер конечного продукта;
- Вечные холивары: юзать Boost в своём проекте или нет (да, юзать);
- Более 100 библиотек, позволяющих легко и быстро заниматься научными расчетами, параллельным программированием, работой с графиками, юнит-тестированием и, конечно же, написанием парсеров!

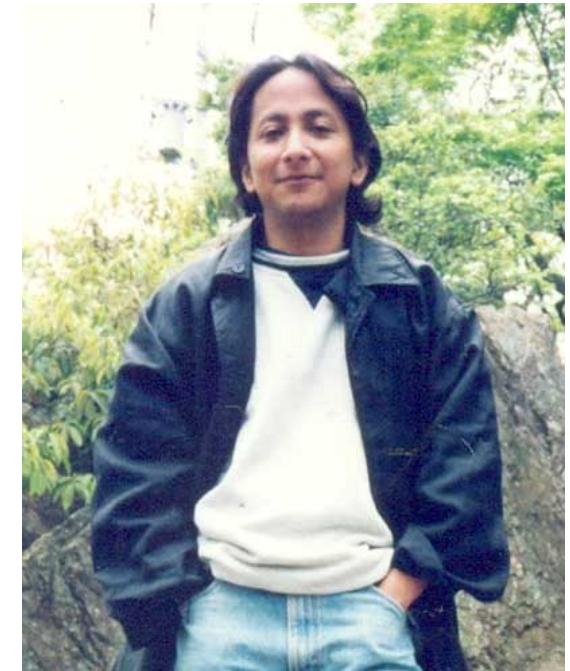


Когда статически слинялся со всем бустом и положил гитхаб

Spirit Parser Framework

90-ые: начало разработки Джоэлом так называемого pre-Spirit. Он был реализован с использованием полиморфных классов времени выполнения. Парсер генерировался в рантайме путем подачи ему строк-правил следующего вида: "prod ::= {'A' | 'B'} 'C';"

2001 – 2006: версия 1.0 - 1.8 была на шаблонах этапа компиляции – static-Spirit. Присоединение к проекту Хартмута Кайзера. Дух официально начал служить в армии Boost'a в 2002 году.



Joel de Guzman



2007: Blazer, эмо, Spirit 2.0.



2016: Boost 1.61.0 & Spirit 2.5.2.

Что мы можем найти в Spirit'e?

- Spirit::Classic – дух для истинных олдфагов. То, чем был Spirit вплоть до версии 1.8. Не рекомендуется к использованию, так как устарел;
- Spirit::Qi – современный Spirit. Именно его сейчас в основном и используют;
- Spirit::Karma – Qi наоборот. Приводит распарсенные данные в удобочитаемый строковый вид;
- Spirit::Lex - генератор лексических анализаторов. Тащем-та не нужен, адназначна.

(целесообразность выделения
отдельного лексического анализатора
иногда находится под вопросом).

А ещё есть Boost::Metaparse,
полностью копирующий Spirit, но
парсящий не в рантайме, а на этапе
компиляции. Впрочем, это уже
совсем другая история...



Много библиотек всяких разных... а я один

Уместить в одну презентацию всё выше перечисленное не представляется возможным, поэтому в дальнейшем мы остановимся подробнее на **Boost::Spirit::Qi**. Её особенности:

- Библиотека способна генерировать нисходящие парсеры с заданием правил распознавания и вывода через формальные грамматики в формате, близком к РБНФ;
- Оформлена в виде DSEL'ов (Domain Specific Embedded Language) — языков, достаточно близких по своему виду к РБНФ-нотации, построенных на конструкциях языка C++ (а именно: шаблонах), что позволяет строить грамматики в удобном виде, так сказать, «не отходя от кассы», прямо в коде C++, без дополнительных шагов по интеграции этого дела с существующим кодом.



РБНФ

РБНФ (Расширенная Форма Бэкуса-Наура) - формальная система определения синтаксиса, в которой одни синтаксические категории последовательно определяются через другие. Используется для описания контекстно-свободных формальных грамматик.

Предложена Никлаусом Виртом. Является расширенной переработкой форм Бэкуса — Наура, отличается от БНФ более «ёмкими» конструкциями, позволяющими при той же выразительной способности упростить и сократить в объёме описание.

```
<expression> ::= <term> ( ('+' | '-') <term> )*
<term>      ::= <factor> ( ('*' | '/') <factor> )*
<factor>     ::= <число> | '+' <factor> | '-' <factor> | '(' <expression> ')'
```

```
expression = term >> *('+' >> term | '-' >> term );
term       = factor >> *('*' >> factor | '/' >> factor );
factor     = uint_ | '+' >> factor | '-' >> factor | '(' >> expression >> ')';
```

Особенности грамматики Spirit

- в БНФ следование одного символа грамматики за другим обозначается просто пробелом, в qī же мы должны явно указывать что после чего следует с помощью оператора следования >>;
- Также '*' в классической РБНФ нотации — постфиксный оператор, равно как и операторы '+', '?'. Увы, в C++ нет постфиксной звёздочки или плюса, зато есть префиксные, так что эти два оператора стали из постфиксных префиксными. Оператора '?' в C++ тоже нет, вместо него используется унарный минус, то есть выражение "expr?" в РБНФ будет эквивалентно "-expr" в нотации Spirit;
 - Наличие базовых элементарных сущностей

Правила грамматики

- Элементарные парсеры. примерами таких парсеров могут быть, например: `int_`, `double_`, `bool_`, `char_`;
- Правила. Основная часть правил в составных грамматиках оперируют по большей части с другими правилами;
- Грамматики. Грамматика в данном случае работает точно так же, как и обыкновенное правило, просто если мы распознаём правило, то парсинг начинается с самого первого парсера, входящего в состав правила, тогда как при распознавании с помощью грамматики парсинг начинается со стартового нетерминала;
 - Таблицы символов;
 - Токены лексического анализатора `Spirit::Lex`;
 - И тд;



Подключаем Boost::Spirit

- Для начала нам нужно установить Boost. Мы можем собрать его из исходников, или скачать уже собранный из rpm, apt, aur и тд (что и рекомендуется);
- Для каждой компоненты Boost::Spirit::Qi имеется свой заголовочный файл, так, например, для функций parse, phrase_parse есть заголовок "boost/spirit/include/qi_parse.hpp", для грамматик — "boost/spirit/include/qi_grammar.hpp", для операторов — "boost/spirit/include/qi_operator.hpp" и так далее. Хоть включение лишь необходимых заголовков сделало бы картину более понятной и хорошо повлияло бы на время компиляции, мы всё же пойдем по лёгкому пути, просто будем подключать заголовок "boost/spirit/include/qi.hpp", который автоматически подключит всё сразу, дабы не отвлекаться на всякие мелочи.



Плюсы и минусы

+

- Позволяет сделать встроенный парсер простеньких данных (писать на духе парсер целого ЯПа – боль);
- Хорошая читабельность парсера (всё же C++ +/- читаемый ЯП);
- Скорость работы;
- Есть исходный код (вряд ли его кто-то будет читать, но так спокойнее);

-

- Сложность отладки (прострелили себе ногу);
- Быстро увеличивается время компиляции с увеличением числа правил грамматики (а если говорить про Metaparser – то всё ещё хуже...)

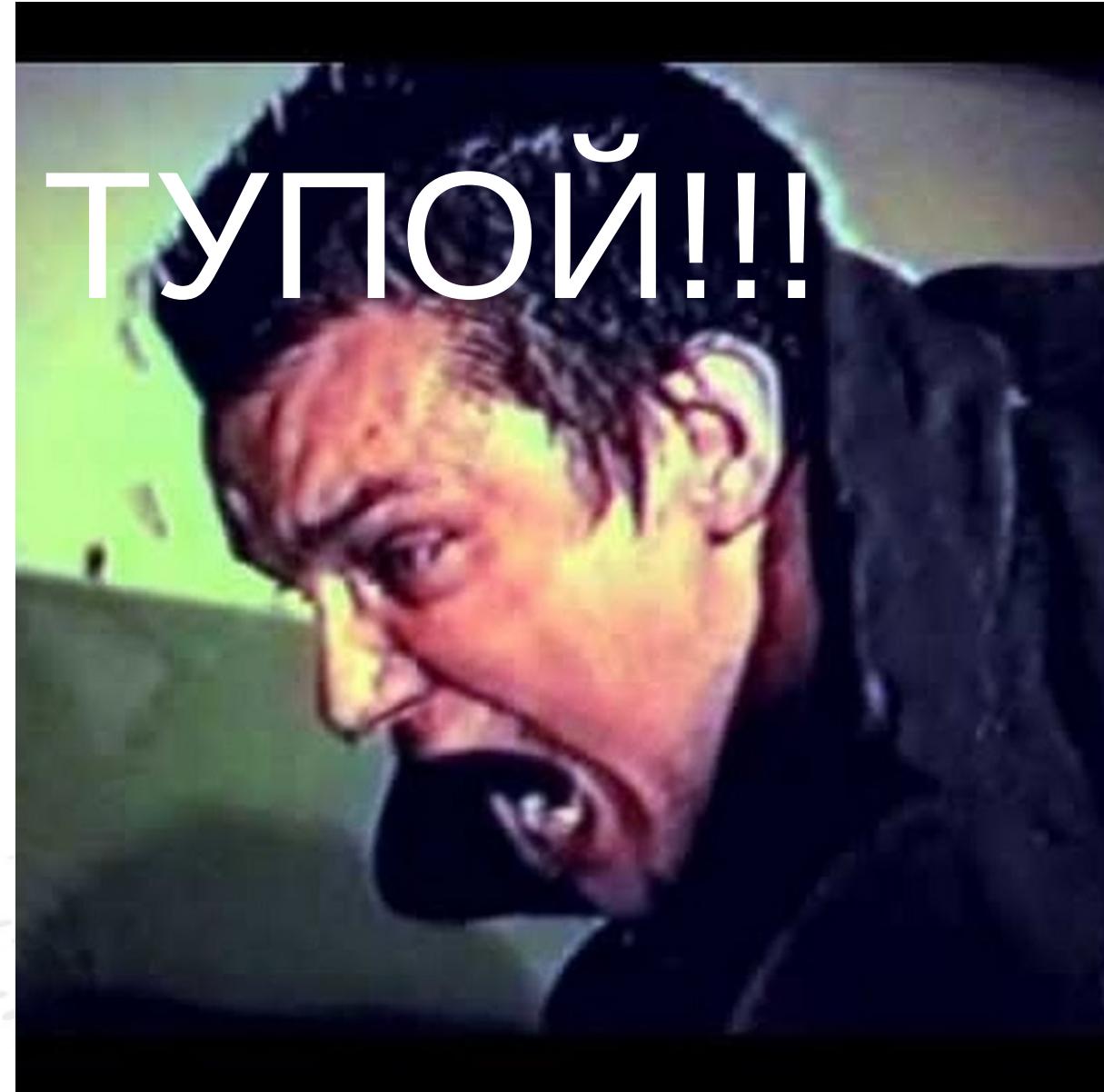
Слишком простой калькулятор...

```
1. //////////////////////////////////////////////////////////////////
2. // calculator_grammar_simple.hpp
3. #ifndef __CALCULATOR_GRAMMAR_SIMPLE_HPP__
4. #define __CALCULATOR_GRAMMAR_SIMPLE_HPP__
5.
6. #ifdef _MSC_VER
7. #pragma once
8. #endif
9.
10. #include <boost/spirit/include/qi.hpp>
11.
12. namespace qi = boost::spirit::qi;
13.
14. template <typename Iterator>
15. struct calculator_simple : qi::grammar<Iterator>
16. {
17.     calculator_simple()
18.         : calculator_simple::base_type(expression)
19.     {
20.         expression = term   >> *( '+' >> term   | '-' >> term );
21.
22.         term      = factor >> *( '*' >> factor | '/' >> factor );
23.
24.         factor    = qi::uint_
25.                   | '(' >> expression >> ')'
26.                   | '+' >> factor
27.                   | '-' >> factor;
28.     }
29.
30.     qi::rule<Iterator> expression, term, factor;
31. };
32.
33. #endif
```

```
1. //////////////////////////////////////////////////////////////////
2. // main.cpp
3. #include <iostream>
4. #include <string>
5.
6. #include <boost/spirit/include/qi.hpp>
7.
8. #include "calculator_grammar_simple.hpp"
9.
10. int main()
11. {
12.     std::cout << "Welcome to the expression parser!\n\n";
13.     std::cout << "Type an expression or [q or Q] to quit\n\n";
14.
15.     typedef std::string      str_t;
16.     typedef str_t::iterator str_t_it;
17.
18.     str_t expression;
19.
20.     calculator_simple<str_t_it> calc;
21.
22.     while(true)
23.     {
24.         std::getline(std::cin, expression);
25.         if(expression == "q" || expression == "Q") break;
26.         str_t_it begin = expression.begin(), end = expression.end();
27.
28.         bool success = qi::parse(begin, end, calc);
29.
30.         std::cout << "-----\n";
31.         if(success && begin == end)
32.             std::cout << "Parsing succeeded\n";
33.         else
34.             std::cout << "Parsing failed\nstopped at: "
35.                         << str_t(begin, end) << "\n";
36.         std::cout << "-----\n";
37.     }
38. }
```

Что не так с нашим калькулятором?

ОН ТУПОЙ!!!



Атрибуты и семантические действия.

- $RT(A_1, \dots, A_N)$, где: RT — тот атрибут, который в результате возвращает само правило; A_1, \dots, A_N — атрибуты-параметры;
- Синтезируемые атрибуты — это то, что возвращает нам успешно отработавший парсер (например, $qi::_val$ — синтезируемый атрибут левой части правила, а $qi::_1$ — атрибут, содержащий результат работы парсера);
 - Каждому парсеру Спирита возможно назначить семантическое действие, которое будет выполнено сразу же, как только успешно отработает парсер, к которому оно присоединено. Запись такая: $P[F]$, где: P — парсер, F — семантическое действие, присоединённое к парсеру P .

```

1. #ifndef __CALCULATOR_GRAMMAR_INTERPRETER_HPP__
2. #define __CALCULATOR_GRAMMAR_INTERPRETER_HPP__
3.
4. #include <boost/spirit/include/qi.hpp>
5.
6. namespace qi = boost::spirit::qi;
7. namespace spirit = boost::spirit;
8.
9. template <typename Iterator>
10. struct calculator_interpreter
11.   : qi::grammar<Iterator, int(), qi::space_type>
12. {
13.     calculator_interpreter()
14.       : calculator_interpreter::base_type(expr)
15.     {
16.       expr =
17.         term [ qi::_val = qi::_1 ]
18.         >> *( '+' >> term [ qi::_val += qi::_1 ]
19.               | '-' >> term [ qi::_val -= qi::_1 ]
20.               )
21.         ;
22.
23.       term =
24.         factor [ qi::_val = qi::_1 ]
25.         >> *( '*' >> factor [ qi::_val *= qi::_1 ]
26.               | '/' >> factor [ qi::_val /= qi::_1 ]
27.               )
28.         ;
29.
30.       factor =
31.         qi::uint_
32.         | '(' >> expr [ qi::_val = qi::_1 ] >> ')'
33.         | '-' >> factor [ qi::_val = -qi::_1 ]
34.         | '+' >> factor [ qi::_val = qi::_1 ]
35.         ;
36.
37.     }
38.
39.     qi::rule<Iterator, int(), qi::space_type>
40.     expr, term, factor;
41. };
42.
43. #endif

```

```

1. #include <iostream>
2. #include <string>
3.
4. #include <boost/spirit/include/qi.hpp>
5.
6. #include "calculator_grammar_interpreter.hpp"
7.
8. int main()
9. {
10.   std::cout << "/////////////////////////////\n\n";
11.   std::cout << "Expression parser...\n";
12.   std::cout << "/////////////////////////////\n\n";
13.   std::cout << "Type an expression... or [q or Q] to quit\n\n";
14.
15.   std::string expression;
16.
17.   calculator_interpreter<std::string::iterator> calc;
18.
19.   while(true)
20.   {
21.     std::getline(std::cin, expression);
22.     if(expression == "q" || expression == "Q") break;
23.     std::string::iterator begin = expression.begin()
24.                           , end = expression.end();
25.
26.     int result;
27.     bool success = qi::phrase_parse( begin
28.                                     , end
29.                                     , calc
30.                                     , qi::space
31.                                     , result);
32.
33.     std::cout << "-----\n";
34.     if(success && begin == end)
35.     {
36.       std::cout << "Parsing succeeded\n";
37.       std::cout << "result = " << result << "\n";
38.     }
39.     else
40.     {
41.       std::cout << "Parsing failed\nstopped at: \""
42.                         << std::string(begin,end) << "\"\n";
43.     }
44.   }
45.   std::cout << "-----\n";

```

Спасибо за внимание!

