

Szablony wyrażeń i Boost Spirit

Romuald Juchnowicz-Bierbasz

Warsaw C++ Users, 2015

Plan prezentacji

Szablony wyrażeń

Wprowadzenie

Implementacja

Właściwości i zastosowania

Boost spirit

Wprowadzenie

Struktura biblioteki

Teoria

Podstawy

Atrybuty parsera

Akcje semantyczne

Pozostałe funkcjonalności

Zalety

Źródła

Problem - Biblioteka do algebry liniowej

Przykładowa implementacja klasy vector:

```
1 #include <vector>
2
3 class vector {
4 public:
5     vector(size_t size) : _data(size) {}
6     double& operator [] (size_t i) {return _data[i];}
7     const double & operator [] (size_t i) const {return _data[i];}
8     size_t size(void) const {return _data.size();}
9 private:
10    std::vector< double> _data;
11 };
```

Chcielibyśmy umożliwić zapis następujących wyrażeń:

```
1 vector v1(3);
2 v1[0] = 0; v1[1] = 1; v1[2] = 2;
3 vector v2(3);
4 v2[0] = 0; v2[1] = 1; v2[2] = 2;
5 vector v3 = v1 + v2 * 2;
```

Pierwsze podejście - przeciążenie operatorów +, *:

```
1 class vector {
2 public:
3     vector(size_t size) : _data(size) {}
4     double& operator [] (size_t i) {return _data[i];}
5     double operator [] (size_t i) const {return _data[i];}
6     size_t size(void) const {return _data.size();}
7     vector operator+(const vector& rhv);
8     vector operator*(double rhv);
9 private:
10     std::vector< double> _data;
11 };
12
13 // jesli chcemy pozwolic na wyrażenia 2 * v to również:
14 vector operator*(double rhv, const vector& lhs);
```

Ile razy zostanie wywołany konstruktor w wyrażeniu:

```
1 vector v3 = v1 + v2 * 2;
```

Ile razy zostanie wywołany konstruktor w wyrażeniu:

```
1 vector v3 = v1 + v2 * 2;
```

Przynajmniej 2 razy (przy założeniu RVO):

- ▶ $\text{tmp} = \text{v2} * 2$
- ▶ $\text{v3} = \text{v1} + \text{tmp}$

Ile razy zostanie wywołany konstruktor w wyrażeniu:

```
1 vector v3 = v1 + v2 * 2;
```

Przynajmniej 2 razy (przy założeniu RVO):

- ▶ $\text{tmp} = \text{v2} * 2$
- ▶ $\text{v3} = \text{v1} + \text{tmp}$

W przypadku bardziej złożonych wyrażeń odpowiednio więcej (5):

```
1 vector v = v1 + v2 * 2 + v3 * 3;
```

A wystarczyłoby jedno wywołanie:

```
1 vector v3(3);  
2 for(size_t i = 0; i < v3.size(); ++i)  
3     v3[i] = v1[i] + v2[i] * 2;
```


A wystarczyłoby jedno wywołanie:

```
1 vector v3(3);  
2 for(size_t i = 0; i < v3.size(); ++i)  
3     v3[i] = v1[i] + v2[i] * 2;
```

Chcielibyśmy, aby poniższe wyrażenie generowało kod równoważny powyższemu:

```
1 vector v3 = v1 + v2 * 2
```

Pierwsza implementacja

```
1  template <typename E1, typename E2>
2  class vector_addition {
3  public:
4      vector_addition( const E1& u, const E2& v)
5          : _u(u), _v(v)
6      {
7          assert(u.size() == v.size());
8      }
9      double operator[](size_t i) const {
10         return _u[i] + _v[i];
11     }
12     size_t size( void) const {return _u.size();}
13 private:
14     const E1& _u;
15     const E2& _v;
16 };
17
18 template <typename E>
19 class vector_scaled {
20 public:
21     vector_scaled( const E& v, double scalar)
22         : _v(v), _scalar(scalar) {}
23     double operator[](size_t i) const {
24         return _v[i] * _scalar;
25     }
26     size_t size( void) const {return _v.size();}
27 private:
28     const E& _v;
29     double _scalar;
30 };
```

Dodajmy odpowiedni konstruktor do klasy vector:

```
1 class vector {
2     ...
3     template <typename E>
4     vector(E const& exp) {
5         _data.resize(exp.size());
6         for (size_type i = 0; i != exp.size(); ++i) {
7             _data[i] = exp[i];
8         }
9     }
10    ...
11};
```

Dodajmy jeszcze operatory, które jednocześnie będą pełniły rolę *funkcji tworzących* (*Creator Functions* - analogicznie do, np. `std::make_pair`):

```
1  template <typename E1, typename E2>
2  vector_addition<E1, E2> operator+(const E1& u, const E2& v)
3  {
4      return vector_addition<E1, E2>(u, v);
5  }
6
7  template <typename E>
8  vector_scaled<E> operator*(const E& v, double scalar)
9  {
10     return vector_scaled<E>(v, scalar);
11 }
```

Możemy teraz zapisać:

```
1  v1 + v2 * 2;
```

Dodajmy jeszcze operatory, które jednocześnie będą pełniły rolę *funkcji tworzących* (*Creator Functions* - analogicznie do, np. `std::make_pair`):

```
1  template <typename E1, typename E2>
2  vector_addition<E1, E2> operator+(const E1& u, const E2& v)
3  {
4      return vector_addition<E1, E2>(u, v);
5  }
6
7  template <typename E>
8  vector_scaled<E> operator*(const E& v, double scalar)
9  {
10     return vector_scaled<E>(v, scalar);
11 }
```

Możemy teraz zapisać:

```
1  v1 + v2 * 2;
```

Jaki typ będzie miało to wyrażenie?

Do wykrywania typu użyjemy sztuczki zaprezentowanej przez Scotta Meyersa w "Effective Modern C++":

```
1 template<typename T>
2 class TD; // deklaracja bez definicji
3
4 TD<decltype(v1 + v2 * 2)> eType;
```

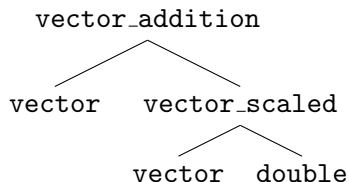
Błąd kompilatora:

```
1>c:\users\romeo\documents\visual studio 2010\projects\spirit\spirit\main.cpp
   (69): error C2079: 'eType' uses undefined class 'TD<T>'
1>         with
1>         [
1>         T=vector_addition<vector ,vector_scaled<vector>>
1>         ]
1>
```

Drzewo składniowe

Nasz typ to:

```
1 vector_addition<  
2   vector,  
3   vector_scaled<  
4     vector  
5   >  
6 >
```



Czy widać tu podobieństwo do znanego wzorca projektowego?

Diagram klas

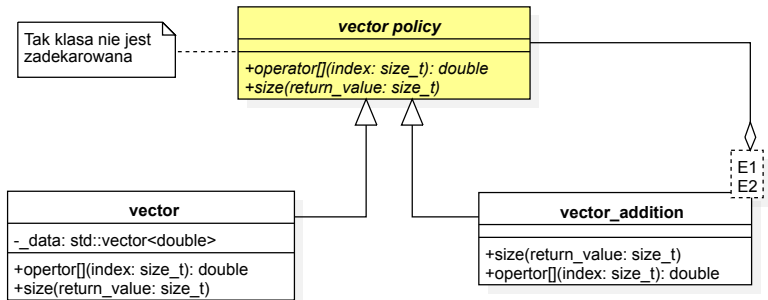


Diagram klas - wzorzec kompozyt

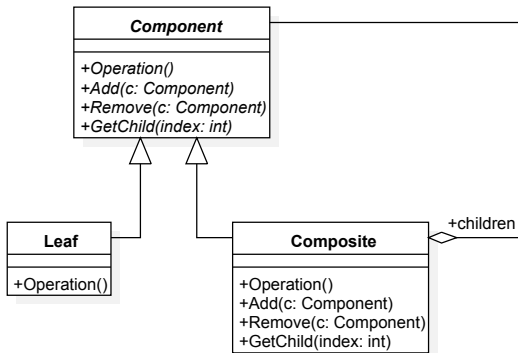
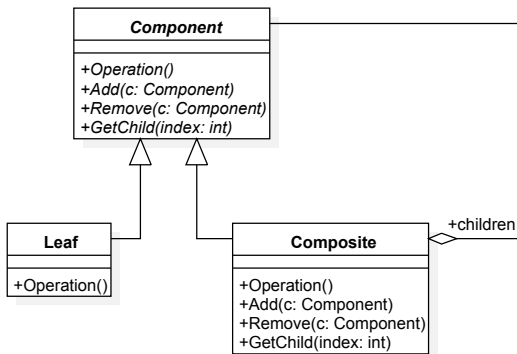


Diagram klas - wzorzec kompozyt



Dlaczego więc po prostu nie stosować wzorca kompozyt?

Pytanie

Are “inline virtual” member functions ever actually “inlined”?

— C++ Super-FAQ

Pytanie

Are “inline virtual” member functions ever actually “inlined”?

— C++ Super-FAQ

Odpowiedź

Occasionally. . .

When the object is referenced via a pointer or a reference, a call to a virtual function cannot be inlined, since the call must be resolved dynamically. Reason: the compiler can't know which actual code to call until run-time (i.e., dynamically), since the code may be from a derived class that was created after the caller was compiled.

Therefore the only time an inline virtual call can be inlined is when the compiler knows the “exact class” of the object which is the target of the virtual function call. This can happen only when the compiler has an actual object rather than a pointer or reference to an object. I.e., either with a local object, a global/static object, or a fully contained object inside a composite.

Szablony wyrażeń pozwalają kompilatorowi na optymalizację kodu poprzez "inline'owanie" metod klas składających się na wyrażenie. Możliwe jest więc aby polecenie:

```
1 vector v = v1 + v2 * 2;
```

Wygenerowało kod analogiczny do:

```
1 vector v3(3);  
2 for(size_t i = 0; i < v3.size(); ++i)  
3     v3[i] = v1[i] + v2[i] * 2;
```

Problem - przeciążanie operatorów

W naszym przykładzie przeciążyliśmy operatory +, *:

```
1  template <typename E1, typename E2>
2  vector_addition<E1, E2> operator+(const E1& u, const E2& v)
3  {
4      return vector_addition<E1, E2>(u, v);
5  }
6
7  template <typename E>
8  vector_scaled<E> operator*(const E& v, double scalar)
9  {
10     return vector_scaled<E>(v, scalar);
11 }
```

Czy taki kod może powodować problemy?

Przeziżyliśmy operatory dla dowolnych typów.

Przeziążyliśmy operatory dla dowolnych typów.

Możemy sobie z tym poradzić dzięki:

- ▶ opakowaniu wszystkiego w przestrzeń nazw,
- ▶ zastosowaniu statycznego polimorfizmu - *Curiously recurring template pattern* - *CRTP*

Curiously recurring template pattern

Przypomnijmy:

```
1 // CRTP
2 template<class T>
3 class Base
4 {
5     // metody klasy Base moga miec dostep do metod/atributow klasy Derived
6 };
7 class Derived : public Base<Derived>
8 {
9     // ...
10};
```

Tworzymy szablonową klasę bazową:

```
1  template <typename E>
2  class expression {
3  public:
4      size_t size() const {return static_cast<E const*>(*this).size();}
5      double operator[](size_t i) const {
6          return static_cast<E const*>(*this)[i];
7      }
8
9      operator E&() {return static_cast<E&>(*this);}
10     operator const E&() const {return static_cast<const E&>(*this);}
11 };
```

Tworzymy szablonową klasę bazową:

```
1 template <typename E>
2 class expression {
3 public:
4     size_t size() const {return static_cast<E const*>(*this).size();}
5     double operator[](size_t i) const {
6         return static_cast<E const*>(*this)[i];
7     }
8
9     operator E&() {return static_cast<E*>(*this);}
10    operator const E&() const {return static_cast<const E*>(*this);}
11};
```

Klasy `vector`, `vector_addition` oraz `vector_scaled` dziedziczą z `expression`:

```
1 class vector : public expression<vector> {
2 public:
3     vector(size_t size) : _data(size) {}
4     template <typename E> vector(expression<E> const& exp);
5     double& operator[](size_t i) {return _data[i];}
6     double operator[](size_t i) const {return _data[i];}
7     size_t size(void) const {return _data.size();}
8 private:
9     std::vector<double> _data;
10};
```

```

1  template <typename E1, typename E2>
2  class vector_addition : public expression<vector_addition<E1, E2> > {
3  public:
4      vector_addition(const E1& u, const E2& v) : _u(u), _v(v) {
5          assert(u.size() == v.size());
6      }
7      double operator[](size_t i) const {return _u[i] + _v[i];}
8      size_t size(void) const {return _u.size();}
9  private:
10     const E1& _u;
11     const E2& _v;
12 };
13
14 template <typename E>
15 class vector_scaled : public expression<vector_scaled<E> > {
16 public:
17     vector_scaled(const E& v, double scalar) : _v(v), _scalar(scalar) {}
18     double operator [] (size_t i) const {return _v[i] * _scalar;}
19     size_t size(void) const {return _v.size();}
20 private:
21     const E& _v;
22     double _scalar;
23 };

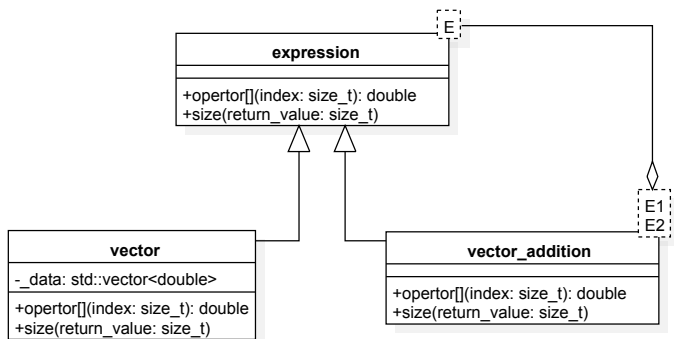
```

Teraz można operatory zapisać jako:

```
1  template <typename E1, typename E2>
2  vector_addition<E1, E2> operator+(
3      const expression<E1>& u, const expression<E2>& v
4  )
5  {
6      return vector_addition<E1, E2>(u, v);
7  }
8
9  template <typename E>
10 vector_scaled<E> operator*(const expression<E>& v, double scalar)
11 {
12     return vector_scaled<E>(v, scalar);
13 }
```

Diagram klas

Uzyskany digram klas jeszcze bardziej przypomina wzorzec kompozyt:



Właściwości i zastosowania szablonów wyrażeń

Właściwości:

- ▶ pozwalają wygenerować szybki kod wynikowy przy pomocy intuicyjnej składni,
- ▶ wydłużają kompilację.

Właściwości i zastosowania szablonów wyrażeń

Właściwości:

- ▶ pozwalają wygenerować szybki kod wynikowy przy pomocy intuicyjnej składni,
- ▶ wydłużają kompilację.

Zastosowania:

- ▶ do tworzenia wbudowanych języków dziedzinowych (*domain-specific embedded language - DSEL*), np.:
 - ▶ nasza przykładowa biblioteka do algebry liniowej,
 - ▶ Blitz++ - obliczenia naukowe,
 - ▶ Boost Spirit - gramatyki EBNF,
- ▶ implementacja wartościowania leniwego (*lazy evaluation*),
- ▶ do łatwej generacji funktorów.

Przykładowe rozwiązanie pozwalające na generację funkcji jednoargumentowych:

```
1  struct variable {
2      double operator() (double v) {return v;}
3  };
4
5  struct constant {
6      double c;
7      constant (double d) : c (d) {}
8      double operator() (double) {return c;}
9  };
10
11 template<class L, class H, class OP>
12 struct binary_expression {
13     L l_;
14     H h_;
15     binary_expression(L l, H h) : l_ (l), h_ (h) {}
16     double operator () (double d) { return OP::apply (l_ (d), h_(d));}
17 };
18
19 struct add {
20     static double apply(double l, double h) {return l + h;}
21 };
22
23 template<class E>
24 struct expression {
25     E expr_;
26     expression (E e) : expr_ (e) {}
27     double operator() (double d) {return expr_(d);}
28 };
```

Dodajmy jeszcze operatory i definicje typów:

```
1 template<class A, class B>
2 expression<binary_expression<expression<A>, expression<B>, add> >
3 operator+(expression<A> a, expression<B> b)
4 {
5     typedef binary_expression<expression<A>, expression<B>, add> ExprT;
6     return expression<ExprT>(ExprT(a,b));
7 }
8
9 typedef expression<variable> var;
10 typedef expression<constant> lit;
```

Możemy teraz napisać:

```
1 var x((variable())); // dlaczego nawiasy?
2 lit l(constant(5.0));
3 double result = (x + l + x)(2); // 9
```

Boost spirit

Wprowadzenie

Problem

Parsowanie listy liczb całkowitych:

"1,2,3,5,12,..." \longrightarrow `std::vector<int>`

Jakie pomysły?

Boost spirit

Wprowadzenie

Problem

Parsowanie listy liczb całkowitych:

"1,2,3,5,12,..." → `std::vector<int>`

Jakie pomysły?

- ▶ pętla ze `scanf` lub `std::istream::operator>>`?
- ▶ `boost::regex`?
- ▶ `getline` i `boost::split`?

Problem

A co gdy wejście się trochę komplikuje?

"1 + 3, 2 * 2, 44 - 33, ..." → `std::vector<int>`

Problem

A co gdy wejście się trochę komplikuje?

" $1 + 3, 2 * 2, 44 - 33, \dots$ " \longrightarrow `std::vector<int>`

W przypadku prostego wejścia wcześniejsze pomysły wystarczają...

Ale gdy złożoność rośnie należy użyć rozwiązania:

- ▶ ładniejszego,
- ▶ bardziej skalowalnego,
- ▶ bardziej odpornego na błędy,
- ▶ etc.

Znane narzędzia do generacji parserów:

- ▶ ANTLR
- ▶ Bison
- ▶ Yacc

Znane narzędzia do generacji parserów:

- ▶ ANTLR
- ▶ Bison
- ▶ Yacc

Wady:

- ▶ kody pośrednie
- ▶ należy nauczyć się ich stosowania
- ▶ linkowanie bibliotek

Znane narzędzia do generacji parserów:

- ▶ ANTLR
- ▶ Bison
- ▶ Yacc

Wady:

- ▶ kody pośrednie
- ▶ należy nauczyć się ich stosowania
- ▶ linkowanie bibliotek

Może Boost.Spirit?

Struktura biblioteki

Biblioteka Boost Spirit składa się z 3 części:

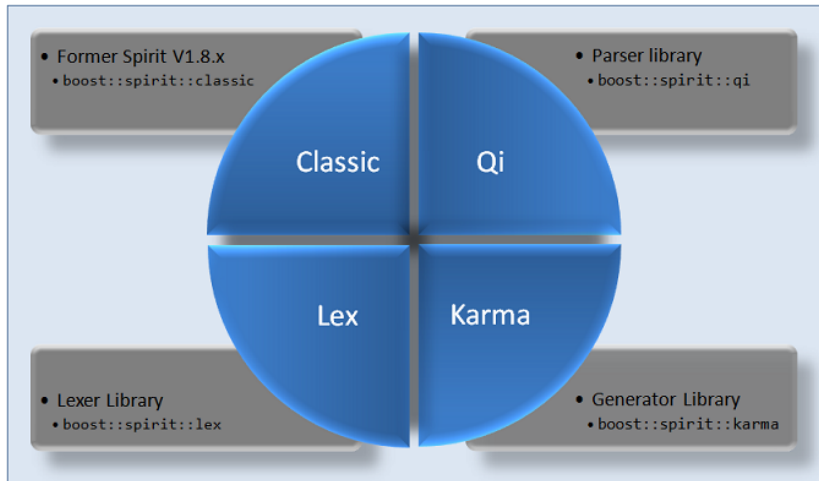
- ▶ lex - do tworzenia lekserów
- ▶ qi - do tworzenia parserów
- ▶ karma - do formatowania danych („odwrotność” qi)

W ramach seminarium dokładniej omówiona zostanie część qi.

Boost spirit zbudowany jest dzięki wykorzystaniu szablonów wyrażeń (przy użyciu Boost Proto). Aby w pełni wykorzystać potencjał biblioteki dobrze jest znać inne biblioteki Boost:

- ▶ Fusion,
- ▶ Phoenix.

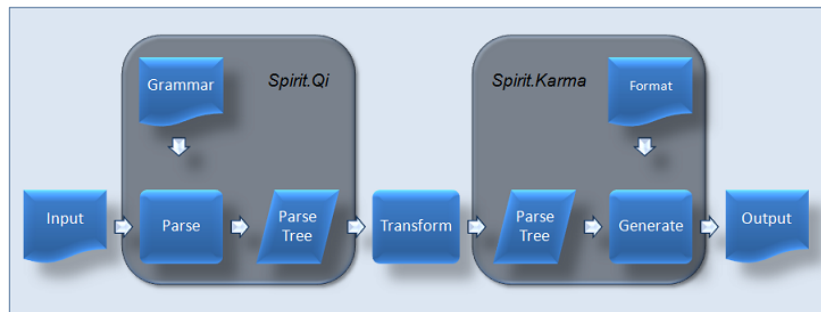
Struktura biblioteki



qi i karma - ying i yang

- ▶ qi zamienia wejście na struktury danych:
`"1|2|3" : string` \longrightarrow `[1, 2, 3] : vector`
- ▶ karma określa formatowanie struktur danych na wyjściu:
`[1, 2, 3] : vector` \longrightarrow `"1|2|3" : string`

Przeptyw danych qi - karma



Gramatyki

Boost Spirit operuje na gramatykach bezkontekstowych przy użyciu rozszerzona notacja Backusa-Naura (*Extended Backus–Naur Form - EBNF*):

- ▶ typ 0 - gramatyka kombinatoryczne,
- ▶ typ 1 - gramatyki kontekstowe,
- ▶ **typ 2 - gramatyki bezkontekstowe,**
- ▶ typ 3 - gramatyki regularne.

Rozszerzona notacja Backusa-Naura

Notacja Backusa-Naura

```
cyfra_bez_zera    ::= "1" | "2" | "3" | "4" | "5"  
                  | "6" | "7" | "8" | "9"  
cyfra             ::= "0" | cyfra_bez_zera  
liczba_naturalna ::= cyfra_bez_zera , { cyfra }  
liczba_calkowita ::= "0" | [ "-" ] , liczba_naturalna
```

Rozszerzona notacja Backusa-Naura

```
cyfra_bez_zera    ::= "1" | "2" | "3" | "4" | "5"  
                  | "6" | "7" | "8" | "9"  
cyfra             ::= "0" | cyfra_bez_zera  
liczba_naturalna ::= cyfra_bez_zera cyfra*  
liczba_calkowita ::= "0" | "-"? liczba_naturalna
```

Qi - podstawy

Boost Spirit Qi pozwala na zdefiniowanie parsera przy pomocy notacji EBNF w kodzie C++:

```
cyfra_bez_zera ::= "1" | "2" | "3" | "4" | "5"  
                | "6" | "7" | "8" | "9"  
cyfra          ::= "0" | cyfra_bez_zera  
liczba_naturalna ::= cyfra_bez_zera cyfra*  
liczba_calkowita ::= "0" | "-"? liczba_naturalna
```

```
1 cyfra_bez_zera = char_('1') | char_('2') | char_('3')  
2               | char_('4') | char_('5')  
3               | char_('6') | char_('7')  
4               | char_('8') | char_('9');  
5 cyfra          = char_('0') | cyfra_bez_zera;  
6 liczba_naturalna = cyfra_bez_zera >> *cyfra;  
7 liczba_calkowita = char_('0') | -char_('-') >> liczba_naturalna;
```


Prosty parser - krok 1

Parser liczby całkowitej:

```
int_
```

Prosty parser - krok 1

Parser liczby całkowitej:

```
int_
```

Qi posiada wiele innych wbudowanych parserów:

- ▶ `double_`
- ▶ `char_`
- ▶ `str`

Prosty parser - krok 2

Parser 2 liczb całkowitych:

```
int_ >> int_
```

Jest to kombinacja dwóch parserów tworząca nowy parser.

Prosty parser - krok 2

Parser 2 liczb całkowitych:

```
int_ >> int_
```

Jest to kombinacja dwóch parserów tworząca nowy parser.

Parser serii liczb całkowitych:

```
*int_
```

Prosty parser - krok 2

Parser 2 liczb całkowitych:

```
int_ >> int_
```

Jest to kombinacja dwóch parserów tworząca nowy parser.

Parser serii liczb całkowitych:

```
*int_
```

Parser serii liczb całkowitych oddzielonych przecinkiem:

```
int_ >> *(char_(' ','') >> int_)
```

Prosty parser - cały kod

```
1 bool parse_numbers(const std::string& s)
2 {
3     using qi::int_;
4     using qi::phrase_parse;
5     using ascii::space;
6
7     std::string::iterator first = s.begin();
8     std::string::iterator last  = s.end();
9
10    bool r = phrase_parse(
11        first,
12        last,
13        int_ >> *(',') >> int_),
14        space // parser białych znaków
15    );
16    if (first != last) // nie udało się sparsować całego wejścia
17        return false;
18    return r;
19 }
```

Prosty parser - cały kod

```
1 bool parse_numbers(const std::string& s)
2 {
3     using qi::int_;
4     using qi::phrase_parse;
5     using ascii::space;
6
7     std::string::iterator first = s.begin();
8     std::string::iterator last  = s.end();
9
10    bool r = phrase_parse(
11        first,
12        last,
13        int_ >> *(',') >> int_),
14        space // parser białych znaków
15    );
16    if (first != last) // nie udało się sparsować całego wejścia
17        return false;
18    return r;
19 }
```

Pytanie

Dlaczego w definicji parsera nie używaliśmy `char_?`

Qi - atrybuty parsera

Każdy parser udostępnia *atrybut syntezowany* - typ „zwracany” po poprawnym parsowaniu, który reprezentują sparsowane wejście. Na przykład:

- ▶ `int_ - int`
- ▶ `double_ - double`
- ▶ `char_ - char`
- ▶ `str - std::string`

Atrybuty złożone

A co ze złożeniem parserów? Przykłady:

Wyrażenie	Typy argumentów	Typ rezultatu
<code>(a >> b)</code>	<code>a: A, b: B</code>	<code>tuple<A, B></code>
<code>(a b)</code>	<code>a: A, b: B</code>	<code>variant<A, B></code>
<code>(a b)</code>	<code>a: A, b: A</code>	<code>A</code>
<code>*a</code>	<code>a: A</code>	<code>vector<A></code>
<code>-a:</code>	<code>a: A</code>	<code>optional<A></code>

Atrybuty złożone - przykłady

Atrybutem syntezowanym

```
int_ >> *(char_(',')) >> int_)
```

jest

```
tuple< int, vector< tuple <char, int> > >
```

Ale już wyrażenie:

```
int_ >> *(lit(',',')) >> int_)
```

ma atrybut syntezowany:

```
vector<int>
```

Akcje semantyczne

Poprzedni parser dawał nam odpowiedź jedynie na pytanie „Czy wejście jest poprawnym wyrażeniem w zadanej gramatyce?”.

To dość mało... Chcemy więcej! Chcemy akcji semantycznych!

Akcje semantyczne

Poprzedni parser dawał nam odpowiedź jedynie na pytanie „Czy wejście jest poprawnym wyrażeniem w zadanej gramatyce?”.

To dość mało... Chcemy więcej! Chcemy akcji semantycznych!

Niech:

- ▶ P - parser
- ▶ A - *akcja*

Korzystając z operatora indeksu możemy przypisać akcję do parsera.

$P[A]$

Akcja A jest wywoływana po poprawnym sparsowaniu wejścia przez parser P.

Akcje semantyczne

Akcję możemy zdefiniować używając:

- ▶ wskaźnika do funkcji
- ▶ obiektu funkcyjnego
- ▶ Boost.Bind i wskaźnika do funkcji
- ▶ Boost.Bind i wskaźnika do metody
- ▶ Boost.Lambda
- ▶ Boost.Phoenix

Akcje semantyczne - przykłady

```
1 namespace qi = boost::spirit::qi;
2 // funkcja
3 void print(int const& i) {
4     std::cout << i << std::endl;
5 }
6
7 // metoda
8 struct writer
9 {
10     void print(int const& i) const {
11         std::cout << i << std::endl;
12     }
13 };
14
15 // funktor
16 struct print_action
17 {
18     void operator()(int const& i, qi::unused_type, qi::unused_type) const {
19         std::cout << i << std::endl;
20     }
21 };
```

Akcje semantyczne - przykłady

Mamy zadane wejście:

```
"{integer}"
```

Chcemy wypisać liczbę znajdującą się w nawiasach:

```
1 // wskaźnik do funkcji
2 parse(first, last, '{' >> int_[&print] >> '}');
3
4 // funktor
5 parse(first, last, '{' >> int_[print_action()] >> '}');
6
7 // Boost.Bind ze wskaźnikiem do funkcji
8 parse(first, last, '{' >> int_[boost::bind(&print, _1)] >> '}');
9
10 // Boost.Bind ze wskaźnikiem do metody
11 writer w;
12 parse(first, last, '{' >> int_[boost::bind(&writer::print, &w, _1)] >> '}');
13
14 // Boost.Lambda
15 parse(first, last, '{' >> int_[std::cout << _1 << '\n'] >> '}');
```

Przykład - parser liczb zespolonych

Parser liczb zespolonych:

'(' >> double_ >> -(',' >> double_) >> ')' | double_

```
1  template <typename Iterator>
2  bool parse_complex(Iterator first, Iterator last, std::complex<double>& c)
3  {
4      /* using boost::: ... */
5
6      double rN = 0.0;
7      double iN = 0.0;
8      bool r = phrase_parse(first, last,
9          // Begin grammar
10         (
11             '(' >> double_[ref(rN) = _1]
12             >> -(',' >> double_[ref(iN) = _1]) >> ')',
13             | double_[ref(rN) = _1]
14         ),
15         // End grammar
16         space);
17
18     if (!r || first != last) // fail if we did not get a full match
19         return false;
20     c = std::complex<double>(rN, iN);
21     return r;
22 }
```


Magia Boost Phoenix

Zamiast `ref` z Boost Phoenix:

```
double_[ref(rN) = _1]
```

możliśmy napisać:

```
1 struct set_double {
2     double& _d;
3     void operator()(double const& d, qi::unused_type, qi::unused_type) const
4     {
5         _d = d;
6     }
7     set_double(double& d) : _d(d) {}
8 };
9
10 ...
11 double_[set_double(nR)]
12 ...
```

Przykład - lista liczb zmiennoprzecinkowych

Zamiast:

```
double_ >> *(',') >> double_)
```

możemy napisać:

```
double_ % ','
```

```
1  template <typename Iterator>
2  bool parse_numbers(Iterator first, Iterator last, std::vector<double>& v)
3  {
4      /* using ... */
5
6      bool r = phrase_parse(first, last,
7
8          // Begin grammar
9          (
10         double_[push_back(ref(v), _1)] % ',','
11         )
12         ,
13         // End grammar
14
15         space);
16
17     if (first != last) // fail if we did not get a full match
18         return false;
19     return r;
20 }
```

Przykład - lista liczb zmiennoprzecinkowych

Możemy od razu otrzymać atrybut syntezowany z parsera:

```
1 template <typename Iterator>
2 bool parse_numbers(Iterator first, Iterator last, std::vector<double>& v)
3 {
4     using qi::double_;
5     using qi::phrase_parse;
6     using qi::_1;
7     using ascii::space;
8
9     bool r = phrase_parse(first, last,
10
11         // Begin grammar
12         (
13             double_ % ','
14         )
15         ,
16         // End grammar
17         space, v);
18
19
20     if (first != last) // fail if we did not get a full match
21         return false;
22     return r;
23 }
```

Pozostałe funkcjonalności

- ▶ definiowanie reguł gramatycznych
- ▶ definiowanie gramatyk
- ▶ struktury jako *atrybut syntezy* (Boost.Fusion)
- ▶ atrybuty dziedziczone
- ▶ obsługa błędów




Zalety Boost Spirit

- ▶ korzysta ze składni C++
- ▶ wyłącznie pliki nagłówkowe
- ▶ bardzo szybka

Źródła I

-  [Expression templates](http://en.wikipedia.org/wiki/Expression_templates)
`http://en.wikipedia.org/wiki/Expression_templates`
-  [More C++ Idioms/Expression-template](http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Expression-template)
`http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Expression-template`
-  [C++ Expression Templates](http://www.angelikalanger.com/Articles/Cuj/ExpressionTemplates/ExpressionTemplates.htm)
`http://www.angelikalanger.com/Articles/Cuj/ExpressionTemplates/ExpressionTemplates.htm`
-  [Boost Spirit homepage](http://boost-spirit.com/home/)
`http://boost-spirit.com/home/`
-  [C++ Super-FAQ](https://isocpp.org/wiki/faq)
`https://isocpp.org/wiki/faq`

Źródła II

-  E. Gamma, R. Helm, R. Johnson, J. Vlissides
Design Patterns
-  S. Meyers
Effective Modern C++
-  A. Alexandrescu
Modern C++ Design