# Introduction à Boost

François-David Collin

16/01/2013

# Introduction

- Importante collection de 117 librairies C++ libres de droit (headers, principalement, sauf pour 12 d'entre elles, à compiler)

# Introduction

- Importante collection de 117 librairies C++ libres de droit (headers, principalement, sauf pour 12 d'entre elles, à compiler)
- Code fourni et revu par les membres du comité TR1/TR2 (future norme C++) Objectifs et Directions

# Introduction

- Importante collection de 117 librairies C++ libres de droit (headers, principalement, sauf pour 12 d'entre elles, à compiler)
- Code fourni et revu par les membres du comité TR1/TR2 (future norme C++) Objectifs et Directions
- Fonctionnalités présentes dans **Boost** « transvasées » au fur et à mesure dans la norme

# Objectifs principaux

- *Réutilisabilité* : implémenter des *patterns* et des fonctions plus ou moins spécifiques couramment utilisées et prêtes à l'emploi (ne pas réinventer la roue)

# Objectifs principaux

- *Réutilisabilité* : implémenter des *patterns* et des fonctions plus ou moins spécifiques couramment utilisées et prêtes à l'emploi (ne pas réinventer la roue)
- *Méthodologie* : illustrer et mettre en vigueur un ensemble de *bonnes pratiques* cohérentes qui tendent vers la généricité, la clarté et la puissance expressive maximale

# Catégories principales

- String and text processing
- Containers
- Iterators

# Catégories principales

- String and text processing
- Containers
- Iterators
- Functions objects and high order programming
- Generic Programming
- Template MetaProgramming

# Catégories principales

- String and text processing
- Containers
- Iterators
- Functions objects and high order programming
- Generic Programming
- Template MetaProgramming
- Maths
- Input/Output
- Concurrent programming

# Catégories principales

- String and text processing
- Containers
- Iterators
- Functions objects and high order programming
- Generic Programming
- Template MetaProgramming
- Maths
- Input/Output
- Concurrent programming
- Etc.

# Exemple : *Iterator*

- Simplifie drastiquement la création et la dérivation d'itérateurs

```
struct is_positive_number {
  bool operator()(int x) { return 0 < x; }
};

int main()
{
  int numbers_[] = { 0, -1, 4, -3, 5, 8, -2 };
  const int N = sizeof(numbers_)/sizeof(int);

  typedef int* base_iterator;
  base_iterator numbers(numbers_);

  // Example using make_filter_iterator()
  std::copy(boost::make_filter_iterator<is_positive_number>(numbers, numbers + N),
            boost::make_filter_iterator<is_positive_number>(numbers + N, numbers + N),
            std::ostream_iterator<int>(std::cout, " "));
  std::cout << std::endl;
```

# Exemple : *Iterator*

- Simplifie drastiquement la création et la dérivation d'itérateurs
- fournit des adaptateurs : `filter_iterator`

```
struct is_positive_number {
  bool operator()(int x) { return 0 < x; }
};

int main()
{
  int numbers_[] = { 0, -1, 4, -3, 5, 8, -2 };
  const int N = sizeof(numbers_)/sizeof(int);

  typedef int* base_iterator;
  base_iterator numbers(numbers_);

  // Example using make_filter_iterator()
  std::copy(boost::make_filter_iterator<is_positive_number>(numbers, numbers + N),
            boost::make_filter_iterator<is_positive_number>(numbers + N, numbers + N),
            std::ostream_iterator<int>(std::cout, " "));
  std::cout << std::endl;
```

# Exemple : *Spirit*

*Parser* permettant d'écrire des grammaires EBNF directement en c++.

```
text %= lexeme[+(char_ - ,<,)];
node %= xml | text;

start_tag %=
        ,<,
    >>  !lit(,/,)
    >>  lexeme[+(char_ - ,>,)]
    >>  ,>,;

end_tag =
        "</"
    >>  string(_r1)
    >>  ,>,;

xml %=
        start_tag[_a = _1]
    >>  *node
    >>  end_tag(_a);
```

# Difficultés

- Deux inconvénients majeurs :

# Difficultés

- Deux inconvénients majeurs :
  1. Temps de compilation (en partie résolue par la précompilation des headers)

# Difficultés

- Deux inconvénients majeurs :
  1. Temps de compilation (en partie résolue par la précompilation des headers)
  2. Courbe d'apprentissage longue (erreurs difficiles à dépister)

```
In file included from SpiritTest.cpp:4:
In file included from ./stdafx.h:11:
In file included from /home/fradav/Dev/boost_1_52_0/boost/spirit/include/qi.hpp:16:
In file included from /home/fradav/Dev/boost_1_52_0/boost/spirit/home/qi.hpp:20:
In file included from /home/fradav/Dev/boost_1_52_0/boost/spirit/home/qi/nonterminal.hpp:14:
In file included from /home/fradav/Dev/boost_1_52_0/boost/spirit/home/qi/nonterminal/rule.hpp:35:
/home/fradav/Dev/boost_1_52_0/boost/spirit/home/qi/reference.hpp:43:30: error: no matching member funct
ion for call to 'parse'
            return ref.get().parse(first, last, context, skipper, attr);
                   ~~~~~~~~~~^~~~~
/home/fradav/Dev/boost_1_52_0/boost/spirit/home/qi/parse.hpp:86:42: note: in instantiation of function
template specialization 'boost::spirit::qi::reference<const boost::spirit::qi::rule<char *, std::basic_
string<char> (), boost::spirit::unused_type, boost::spirit::unused_type, boost::spirit::unused_type> >:
:parse<const char *, boost::spirit::context<boost::fusion::cons<std::basic_string<char> &, boost::fusio
n::nil>, boost::spirit::locals<mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mp
l_::na, mpl_::na, mpl_::na> >, boost::spirit::unused_type, std::basic_string<char> >' requested here
        return compile<qi::domain>(expr).parse(first, last, context, unused, attr);
                                         ^
./Header.h:40:9: note: in instantiation of function template specialization 'boost::spirit::qi::parse<c
onst char *, boost::spirit::qi::rule<char *, std::basic_string<char> (), boost::spirit::unused_type, bo
ost::spirit::unused_type, boost::spirit::unused_type>, std::basic_string<char> >' requested here
    if (parse(f, l, p, attr) && (!full_match || (f == l)))
        ^
SpiritTest.cpp:216:5: note: in instantiation of function template specialization 'test_parser_attr<boos
t::spirit::qi::rule<char *, std::basic_string<char> (), boost::spirit::unused_type, boost::spirit::unus
ed_type, boost::spirit::unused_type>, std::basic_string<char> >' requested here
    test_parser_attr("\"truc\"", qstring, res1);
    ^
/home/fradav/Dev/boost_1_52_0/boost/spirit/home/qi/nonterminal/rule.hpp:273:14: note: candidate functio
n [with Context = boost::spirit::context<boost::fusion::cons<std::basic_string<char> &, boost::fusion::
nil>, boost::spirit::locals<mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_:
:na, mpl_::na, mpl_::na> >, Skipper = boost::spirit::unused_type, Attribute = std::basic_string<char>]
not viable: no known conversion from 'const char *' to 'char *&' for 1st argument;
        bool parse(Iterator& first, Iterator const& last
             ^
/home/fradav/Dev/boost_1_52_0/boost/spirit/home/qi/nonterminal/rule.hpp:319:14: note: candidate functio
n template not viable: requires 6 arguments, but 5 were provided
        bool parse(Iterator& first, Iterator const& last
             ^
```

# Difficultés

- Deux inconvénients majeurs :
  1. Temps de compilation (en partie résolue par la précompilation des headers)
  2. Courbe d'apprentissage longue (erreurs difficiles à dépister)
  3. Exemple d'erreur de compilation avec **Spirit**
- Mais...

# ... c'est le futur !

- Tendance : les *DSEL* (*Domain Specific Embedded Language*)

# … c'est le futur !

- Tendance : les *DSEL* (*Domain Specific Embedded Language*)
- Dans cette optique **Boost** est une référence ; en se fondant sur la *méta-programmation par templates* (*MPT*), il offre un environnement consistant pour coder des DSEL

# Conclusion

- Conseil : acquérir quelques rudiments en *MPT*

# Conclusion

- Conseil : acquérir quelques rudiments en *MPT*
- certaines librairies semblent complexes à première vue mais pour beaucoup d'entre elles, il s'agit de concepts très simples.

# Conclusion

- Conseil : acquérir quelques rudiments en *MPT*
- certaines librairies semblent complexes à première vue mais pour beaucoup d'entre elles, il s'agit de concepts très simples.
- **Boost** comme la *MPT* en général, est à double-tranchant :

# Conclusion

- Conseil : acquérir quelques rudiments en *MPT*
- certaines librairies semblent complexes à première vue mais pour beaucoup d'entre elles, il s'agit de concepts très simples.
- **Boost** comme la *MPT* en général, est à double-tranchant :
  - permet de compacter, clarifier et rendre le code générique/adaptable

# Conclusion

- Conseil : acquérir quelques rudiments en *MPT*
- certaines librairies semblent complexes à première vue mais pour beaucoup d'entre elles, il s'agit de concepts très simples.
- **Boost** comme la *MPT* en général, est à double-tranchant :
  - permet de compacter, clarifier et rendre le code générique/adaptable
  - augmente la complexité formelle

# References

- La future source de vos joies et pleurs : Site Boost

# References

- La future source de vos joies et pleurs : Site Boost
- Les meilleures introductions à la méta-programmation :

# References

- La future source de vos joies et pleurs : Site Boost
- Les meilleures introductions à la méta-programmation :
  1. [Alexandrescu 2001]

# References

- La future source de vos joies et pleurs : Site Boost
- Les meilleures introductions à la méta-programmation :
  1. [Alexandrescu 2001]
  2. [Abrahams and Gurtovoy 2004]

# References

- La future source de vos joies et pleurs : Site Boost
- Les meilleures introductions à la méta-programmation :
  1. [Alexandrescu 2001]
  2. [Abrahams and Gurtovoy 2004]
- Introduction directe à **Boost** :

# References

- La future source de vos joies et pleurs : Site Boost
- Les meilleures introductions à la méta-programmation :
  1. [Alexandrescu 2001]
  2. [Abrahams and Gurtovoy 2004]
- Introduction directe à **Boost** :
  1. [Karlsson 2005]

# References

- La future source de vos joies et pleurs : Site Boost
- Les meilleures introductions à la méta-programmation :
  1. [Alexandrescu 2001]
  2. [Abrahams and Gurtovoy 2004]
- Introduction directe à **Boost** :
  1. [Karlsson 2005]
  2. [Schäling 2011]

Abrahams, D. and Gurtovoy, A., 2004. *C++ template metaprogramming : Concepts, tools, and techniques from boost and beyond*, Addison-Wesley Professional.

Alexandrescu, A., 2001. *Modern C++ design : generic programming and design patterns applied*, Addison-Wesley Professional.

Karlsson, B., 2005. *Beyond the C++ Standard Library : An Introduction to Boost*, Addison-Wesley Professional.

Schäling, B., 2011. *The Boost C++ Libraries*, Xml Press.