

# Text Processing With Boost

*Or, "Beating Perl at Its Own Game"*

# Talk Overview

- The Simple Stuff
  - Boost.LexicalCast
  - Boost.StringAlgo
  - Boost.Tokenizer
  - Boost.Format
- The Advanced Stuff
  - Boost.Regex
  - Boost.Xpressive
  - Boost.Spirit
- The Semi-Secret Unicode Stuff

# Hidden Agenda

Declarative Programming and Domain-Specific  
Languages

# Part I: The Simple Stuff

*Utilities for Ad Hoc Text Manipulation*

# A Legacy of Inadequacy

- Python:

```
>>> int('123')
123
>>> str(123)
'123'
```

- C:

```
int i = atoi("123");
```

```
char buff[10];
itoa(123, buff, 10);
```

Not in the standard!

No error handling!

No error handling!

Complex interface



boostpro  
computing

# Using std::stringstream

```
{ // int --> string
    std::stringstream sout;
    std::string str;
    sout << 123;
    sout >> str;    // OK, str == "123"
}

{ // string --> int
    std::stringstream sout;
    int i;
    sout << "789";
    sout >> i;      // OK, i == 789
}
```

# Boost.LexicalCast

```
#include <boost/lexical_cast.hpp>

int i = lexical_cast<int>( "123" );

std::string str = lexical_cast<std::string>( 789 );
```

- ✓ Clean Interface
- ✓ Error Reporting, Yay!
- ✓ Extensible
- ✗ Ugly name
- ✗ Sub-par performance
- ✗ No internationalization

# Boost.LexicalCast

```
// Approximate implementation ...
template< typename Target, typename Source >
Target lexical_cast(Source const & arg)
{
    std::stringstream sout;
    Target result;

    if(!(sout << arg &&
        sout >> result && sout.eof()))
        throw bad_lexical_cast(
            typeid(Source), typeid(Target));

    return result; Was input consumed?
}
```



Kevlin  
Henney  
boostpro  
computing

# LexicalCast Quiz!

```
typedef char int8_t;  
int8_t i = lexical_cast<int8_t>( "42" );
```

What is the value of i?

Answer: throws  
boost::bad\_lexical\_cast;

# More Inadequacy

- From Wikipedia, Trim\_(programming):
  - In programming, **trim** or **strip** is a string manipulation function or algorithm which removes leading and trailing whitespace from a string.
- For example, in Python:

```
>>> '    testy, testy    '.strip  
'testy, testy'  
>>>
```

Lame!

- There is no standard trim function in C or C++.

# Boost.StringAlgo

- Extension to `std::` algorithms
- Generic, works with any string-like thing
- Includes algorithms for:
  - trimming
  - case-conversions
  - find/replace utilities
  - ... and much more!

# Hello, StringAlgo!

```
#include <boost/algorithm/string.hpp>
using namespace std;
using namespace boost;

std::string str1(" hello world! ");
boost::to_upper(str1);           // str1 == " HELLO WORLD! "
boost::trim(str1);              // str1 == "HELLO WORLD!"
```

In-place mutation

```
std::string str2 =
    boost::to_lower_copy(
        boost::ireplace_first_copy(
            str1, "hello", "goodbye")));
// str2 == "goodbye world!"
```

Create a new string

Composable Algorithms!

# Plenty of algo's to choose from!

to_upper_copy	ilexicographical_compare	erase_last	replace_tail_copy	contains
to_upper	all	erase_last_copy	erase_tail	icontains
to_lower_copy	find_first	ierase_last	erase_tail_copy	equals
to_lower	ifind_first	ierase_last_copy	replace_regex	iequals
trim_left_copy_if	find_last	replace_nth	replace_regex_copy	lexicographical_compare
trim_left_if	ifind_last	replace_nth_copy	erase_regex	ierase_first_copy
trim_left_copy	find_nth	ireplace_nth	erase_regex_copy	replace_last
trim_left	ifind_nth	ireplace_nth_copy	replace_all_regex	replace_last_copy
trim_right_copy_if	find_head	erase_nth	replace_all_regex_copy	ireplace_last
trim_right_if	find_tail	erase_nth_copy	erase_all_regex	ireplace_last_copy
trim_right_copy	find_token	ierase_nth	erase_all_regex_copy	replace_head
trim_right	find_regex	ierase_nth_copy	find_format	replace_head_copy
trim_copy_if	find	replace_all	find_format_copy	erase_head
trim_if	replace_first	replace_all_copy	find_format_all	erase_head_copy
trim_copy	replace_first_copy	ireplace_all	find_format_all_copy	replace_tail
trim	ireplace_first	ireplace_all_copy	find_all	join
starts_with	ireplace_first_copy	erase_all	ifind_all	join_if
istarts_with	erase_first	erase_all_copy	find_all_regex	
ends_with	erase_first_copy	ierase_all	split	
iends_with	ierase_first	ierase_all_copy	split_regex	

# StringAlgo: split

```
std::string str( "abc-*ABC-*-aBc" );
std::vector< std::string > tokens;

boost::split( tokens, str, boost::is_any_of("-*") );
// OK, tokens == { "abc", "ABC", "aBc" }
```

Other Classifications:  
is\_space, is\_upper, etc.  
is\_from\_range('a','z'),  
is\_alnum || is\_punct

# Boost.Tokenizer

- Powerful and flexible tools to split strings into tokens
  - Container-like interface, or
  - Iterator interface
- Parsing done lazily

# Hello, Tokenizer!

```
#include <string>
#include <iostream>
#include <boost/tokenizer.hpp>

int main()
{
    std::string s = "This is, a test";
    boost::tokenizer<> tokens(s);
    for (boost::tokenizer<>::iterator beg =
        tokens.begin(); beg != tokens.end(); ++beg)
    {
        std::cout << *beg << "\n";
    }
}
```

Behaves like a lazy container of  
std::string

References string stored in beg

# Hello, Tokenizer!

```
#include <string>
#include <iostream>
#include <boost/tokenizer.hpp>
#include <boost/foreach.hpp>
int main()
{
    std::string s = "This is, a test";
    boost::tokenizer<> tokens(s);
    BOOST_FOREACH(std::string& tok, tokens)
    {
        std::cout << tok << "\n";
    }
}
```

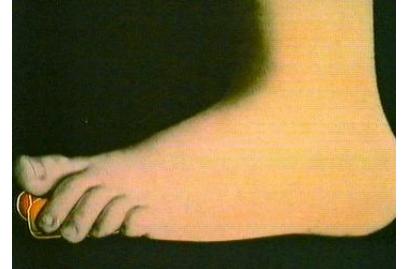
# char\_separator<>

```
#include <string>
#include <iostream>
#include <boost/tokenizer.hpp>
#include <boost/foreach.hpp>
int main()
{
    std::string s = "This-;|is;;;a|-;test";
    boost::char_separator<char> sep("|-;");
    boost::tokenizer<
        boost::char_separator<char> > tokens(s, sep);
    BOOST_FOREACH(std::string tok, tokens)
        std::cout << tok << "\n";
}
```

# Other Tokenizer Functions

- `escaped_list_separator` — specify separator and an escape for literal separators in tokens
- `offset_separator` — break strings at integer offsets
- Define your own — must conform to the `TokenizerFunction` concept, defined in the docs.

# It's funny ... laugh.



"If iostreams are a step towards the future, I sure hope the future will have a definitive solution for the Carpal Tunnel Syndrome."

-- Andrei Alexandrescu, CUJ Aug. 2005

# What's wrong with IOStreams?

```
unsigned int i = 0xffff;

// This prints: "**** 0xffff****65535****"
std::printf("****%#10x****%u****\n", i, i);

// The equivalent using C++ IOStreams ...
std::ios_base::fmtflags f = std::cout.flags();
std::cout << "****"
    << std::showbase
    << std::setw(10)
    << std::hex
    << i;
std::cout.flags(f);
std::cout << "****"
    << i
    << "****\n";
```

# What's wrong with IOStreams?

	printf	iostream
Type-safe	✗	✓
Extensible	✗	✓
Concise	✓	✗
Efficient	✓	✗
Separation of format from data	✓	✗

# Boost.Format

---

- Type-safe printf().
- Stream-based, so works with user-defined types!
- Concise.
- Separation of format string and data.



*Samuel Krempp*

# Hello, Boost.Format

```
unsigned int i = 0xffff;  
  
// This prints: "***      0xffff***65535***"  
std::printf("%%%#10x%%%u***\n", i, i);  
  
// The equivalent using Boost.Format ...  
std::cout << boost::format("%%%#10x%%%u***\n") %i %i;
```

- (Mostly) format compatible with printf() ...
- ... and type-safe, too!

# Hello, Boost.Format

---

- Positional arguments:

```
// prints: "42 hello 42":  
cout << boost::format("%1% %2% %1%") % 42 % "hello";
```

- printf()-style formats:

```
// prints: "(x,y) = (-23, +35)"  
cout << boost::format("(x,y) = (%+5d,%+5d)") % -23 % 35;
```

# Hello, Boost.Format

- You can use IO manipulators, and reuse format obj:

```
// prints: "*  +42*42*"
boost::format f("*%1%*%1%*\n");
f.modify_item(1, boost::io::group(std::showpos, std::setw(5)));
cout << f % 42 ;
```

- Formatting can be used with arguments, too:

```
// prints: "*  +42*  +42*"
cout << boost::format("*%1%*%1%*")
    % boost::io::group(std::showpos, std::setw(5), 42);
```

# Text Formatting Grudge Match

	printf	iostream	format
Type-safe	✗	✓	✓
Extensible	✗	✓	✓
Concise	✓	✗	✓
Efficient	✓	✗	✗
Separation of format from data	✓	✗	✓

# Exercise 1: CSV → Tiny XML

---

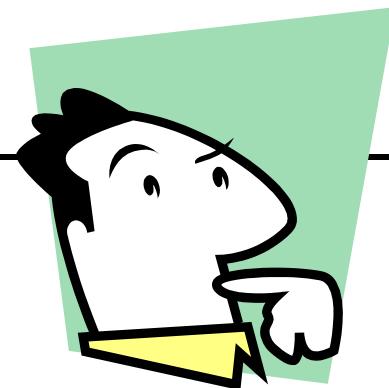
- Use Boost.Tokenizer to parse the input
- Use Boost.StringAlgo to turn column headings into tag names
- Use Boost.Format to write the output

Name, Birthplace, Birth year  
 Homer Simpson, Saskatchewan\*, 1981  
 David Abrahams, Princeton\, NJ, 1964  
 ...



```
<record>
    <name>Homer Simpson</name>
    <birthplace>Saskatchewan*</birthplace>
    <birth_year>1981</birth_year>
</record>
<record>
    <name>David Abrahams</name>
    <birthplace>Princeton, NJ</birthplace>
    <birth_year>1964</birth_year>
</record>
...
```

- Bonus: use Boost.LexicalCast to inject an <age> tag with a value computed from birth\_year



# Exercise 1

## (in progress)

Name, Birthplace, Birth year  
Homer Simpson, Saskatchewan\*,  
1981  
David Abrahams, Princeton\, NJ,  
1964  
...



```
<record>
  <name>Homer Simpson</name>
  <birthplace>Saskatchewan*</birthplace>
  <birth_year>1981</birth_year>
  <age>27</age>
</record>
<record>
  <name>David Abrahams</name>
  <birthplace>Princeton, NJ</birthplace>
  <birth_year>1964</birth_year>
  <age>44</age>
</record>
...
```

# Part 2: The Advanced Stuff

*Structured Text Manipulation with  
Domain Specific Languages*

# Overview

---

- Declarative Programming and Domain-Specific Languages.
- Manipulating Text Dynamically
  - Boost.Regex
- Generating Parsers Statically
  - Boost.Spirit
- Mixed-Mode Pattern Matching
  - Boost.Xpressive

# Grammar Refresher



**Imperative Sentence:** *n.*

Expressing a command or request.

*E.g., “Set the TV on fire.”*

**Declarative Sentence:** *n.*

Serving to declare or state.

*E.g., “The TV is on fire.”*

# Computer Science Refresher

---

## **Imperative Programming:** *n.*

A programming paradigm that describes computation in terms of a *program state* and statements that change the program state.

## **Declarative Programming:** *n.*

A programming paradigm that describes computation in terms of *what* to compute, not *how* to compute it.

# Find/Print an Email Subject

---

```
std::string line;
while(std::getline(std::cin, line))
{
    if(line.compare(0, 9, "Subject: ") == 0)
    {
        std::size_t offset = 9;
        if(line.compare(offset, 4, "Re: ") == 0)
            offset += 4;
        std::cout << line.substr(offset);
    }
}
```

# Find/Print an Email Subject

---

```
std::string line;
boost::regex pat( "^Subject: (Re: )?(.*)" );
boost::smatch what;

while (std::getline(std::cin, line))
{
    if (boost::regex_match(line, what, pat))
        std::cout << what[2];
}
```

# Which do you prefer?

Imperative:

```
if (line.compare(...) == 0)
{
    std::size_t offset = ...;
    if(line.compare(...) == 0)
        offset += ...;
}
```

Declarative:

```
"^Subject: (Re: )?(.*)"
```

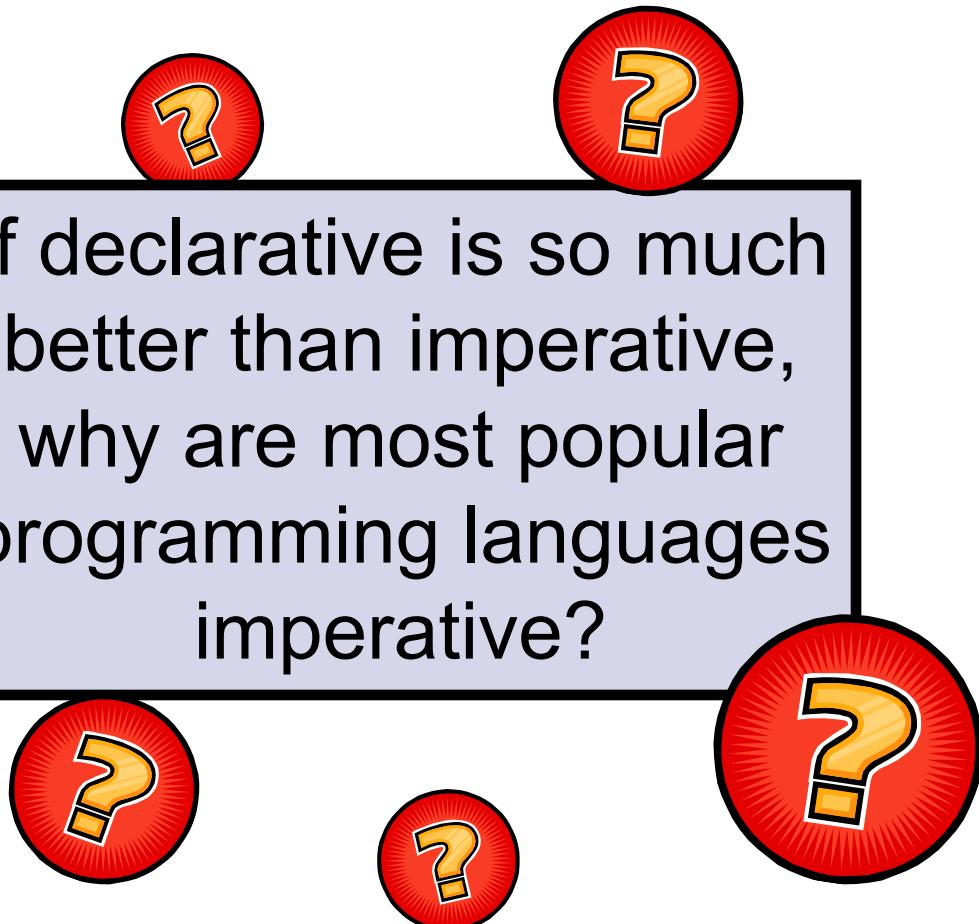
- ✗ Describes algorithm
- ✗ Verbose
- ✗ Hard to maintain

- ✓ Describes goal
- ✓ Concise
- ✓ Easy to maintain

# Riddle me this, Batman...



If declarative is so much better than imperative,  
why are most popular programming languages  
imperative?



# Best of Both Worlds

---

- Domain-Specific *Embedded* Languages
  - A declarative DSL hosted in an imperative general-purpose language.
- Examples:
  - Ruby on Rails in Ruby
  - JUnit Test Framework in Java
  - Regex in perl, C/C++, .NET, etc.

# Boost.Regex in Depth

---

- A powerful DSEL for text manipulation
- Accepted into std::tr1
  - Coming in C++0x!
- Useful constructs for:
  - matching
  - searching
  - replacing
  - tokenizing



*John Maddock*

# Dynamic DSEL in C++

---

- Embedded statements in strings
- Parsed at runtime
- Executed by an interpreter
- Advantages
  - Free-form syntax
  - New statements can be accepted at runtime
- Examples
  - regex: "^\Subject: (Re: )?(.\*)"
  - SQL: "SELECT \* FROM Employees ORDER BY Salary"

# The Regex Language

---

Syntax	Meaning
<code>^</code>	Beginning-of-line assertion
<code>\$</code>	End-of-line assertion
<code>.</code>	Match any single character
<code>[abc]</code>	Match any of 'a', 'b', or 'c'
<code>[^0-9]</code>	Match any character not in the range '0' through '9'
<code>\w, \d, \s</code>	Match a word, digit, or space character
<code>* , + , ?</code>	Zero or more, one or more, or optional (postfix, greedy)
<code>(stuff)</code>	Numbered capture: remember what <i>stuff</i> matches
<code>\1</code>	Match what the 1 <sup>st</sup> numbered capture matched

# Quiz: What Do These Do?

---

"\\d\\d?-\\d\\d?-\\d\\d(\\d\\d)?"

*Match a date, e.g. "5-30-73"*

<(\\w+)>.\*</\\1>"

*Match HTML tags, e.g. "<b>Bold!</b>"*

"\\d{3}-\\d\\d-\\d{4}"

*Match a Social Security Number*

# Algorithm: regex\_match()

- Checks if a pattern matches the whole input.
- Example: Match a Social Security Number

```
std::string line;
boost::regex ssn("\\d{3}-\\d\\d-\\d{4}");

while (std::getline(std::cin, line))
{
    if (boost::regex_match(line, ssn))
        break;
    std::cout << "Invalid SSN. Try again.\n";
}
```

# Algorithm: regex\_search()

- Scans input to find a match
- Example: scan HTML for an email address

```
std::string html = /*...*/ ;
regex mailto("<a href=\"mailto:(.*?)\">",
            regex_constants::icase);
smatch what;

if(boost::regex_search(html, what, mailto))
{
    std::cout << "Email address to spam: " << what[1];
}
```

# Algorithm: regex\_replace()

- Replaces occurrences of a pattern
- Example: Simple URL escaping

```
std::string url("http://foo.net/this has spaces");
std::string format("%20");
boost::regex pat(" ");

// This changes url to
// "http://foo.net/this%20has%20spaces"
url = boost::regex_replace(url, pat, format);
```

# Iterator: regex\_iterator

- Iterates through all occurrences of a pattern
- Example: scan HTML for email addresses

```
std::string html = /*...*/ ;
regex mailto("<a href=\"mailto:(.*?)\">",
            regex_constants::icase);
sregex_iterator begin(html.begin(), html.end(), mailto);
sregex_iterator end;

for(; begin != end; ++begin)
{
    smatch const & what = *begin;
    std::cout << "Email address to spam: " << what[1];
}
```

# Iterator:

## regex\_token\_iterator

- Tokenizes input according to pattern
- Example: scan HTML for email addresses

```
std::string html = /* ... */ ;
regex mailto("<a href=\"mailto:(.*?)\">",
            regex_constants::icase);

sregex_token_iterator begin(html.begin(), html.end(),
                           mailto, 1);
sregex_token_iterator end;

using namespace boost::lambda;
std::for_each(begin, end, std::cout << _1 << '\n');
```

# Know your iterator type

---

- Pick the regex type depending on your iterator

<b>sregex</b>	<b>std::string::const_iterator</b>
cregex	char const *
wsregex	std::wstring::const_iterator
wcregex	wchar_t const *

# Know your task

---

See if a whole string matches	regex_match() algorithm
See if a string contains a sub-string that matches	regex_search() algorithm
Replace all sub-strings that match	regex_replace() algorithm
Iterate through all sub-strings that match	regex_iterator<> class
Split a string into tokens that each match a regex	regex_token_iterator<> class
Split a string using a regex as a delimiter	regex_token_iterator<> class

# Regex examples

---

- Are so well presented in the documentation that we don't need to use PowerPoint



# Exercise 2

## (in progress)

### ■ Use Boost.Regex to

1. Replace all occurrences of numbers by the character '#'
2. Find and sum all integer numbers
3. Split the text into tokens delimited by any sequence of punctuation characters or whitespace
4. Print all dates, converting from US format "2/21/2008" to European format "21. 2. 2008"
5. Change all dates from US format "2/21/2008" to European format "21. 2. 2008"

# Regex Challenge!

- Write a regex to match balanced, nested braces, e.g. "{ foo { bar } baz }"

```
regex braces("{[^{}]*}");
```

Not quite.

```
regex braces("{{[^{}]*({{[^{}]*}}*[^{}]*})*}}");
```

Better, but no.

```
regex braces("{{[^{}]*|{{[^{}]*|{{[^{}]*|{{[^{}]*}}}}}}}}")
```

## Not there, yet.

## regex

# It's funny ... laugh.



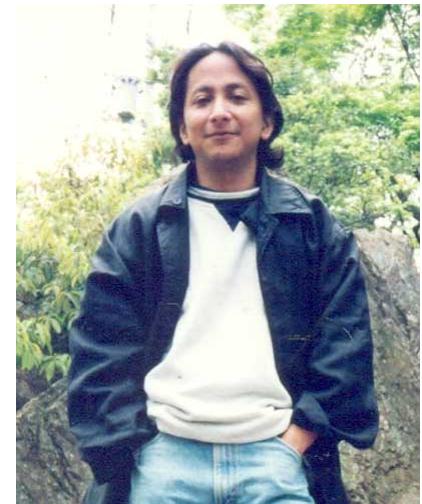
“Some people, when confronted with a problem, think, ‘*I know, I’ll use regular expressions.*’ Now they have two problems.”

--Jamie Zawinski, in alt.religion.emacs

# Introducing Spirit.Qi



- Use v2, not v1!
  - <http://boost.org/libs/spirit/doc/html>
- Parser Generator
  - similar in purpose to lex / YACC
- DSEL for declaring *grammars*
  - grammars can be recursive
  - DSEL approximates Backus-Naur Form
- *Statically* embedded language
  - Domain-specific statements are composed from C++ expressions.



Joel de  
Guzman

---

# Boost.Spirit – The Basics

“Everything’s a Parser.” \*

# Floating-Point Numbers

---

double\_

A parser

(Unless otherwise specified, assume

using namespace boost::spirit;

)

# Two Floating-Point Numbers

---

double\_ >> double\_  


A composition of parsers  
(also a parser)

>> means “followed by”

# Zero or More Floating-Point Numbers

---

\*double\_

You guessed it...  
(another parser)

\**p* means “zero or more instances of *p*.”

# List of Floating-Point Numbers

---

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

parser

char\_('c') means "the character 'c'."

(...) groups expressions as usual

# List of Floating-Point Numbers

---

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

parser

char\_('c') means "the character 'c'."

(...) groups expressions as usual

# List of Floating-Point Numbers

---

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

parser

char\_('c') means "the character 'c'."

(...) groups expressions as usual

# List of Floating-Point Numbers

---

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

parser

char\_('c') means "the character 'c'."

(...) groups expressions as usual

# Actually Parse a Sequence

```
parse( s.begin(), s.end(),
    double_ >> *(char_(',') >> double_))
```

parser

3.14,2.72,99.9

parse(*start, finish, p*) → bool

# Actually Parse a Sequence

```
parse( s.begin(), s.end(),
    double_ >> *(char_(',') >> double_))
```

parser

```
3.14, 2.72, 99.9
```

```
parse(start, finish, p) → bool
```

# Actually Parse a Sequence

```
phrase_parse( s.begin(), s.end(),
    double_ >> *(char_(',') >> double_), space)
```

parser

3.14, 2.72, 99.9

“skip parser”  
(just another parser)

phrase\_parse(*start, finish, p, s*) → bool

# Shorthand

```
phrase_parse( s.begin(), s.end(),
```

```
    double_ % char_(','),
```

```
    space)
```

parser

```
3.14, 2.72, 99.9
```

“skip parser”

(just another parser)

$a \% b$  means  $a >> * (b >> a)$

# Shorthand

```
phrase_parse( s.begin(), s.end(),
```

```
    double_ % ',',
```

```
    space)
```

parser

```
3.14, 2.72, 99.9
```

“skip parser”  
(just another parser)

$a \% b$  means  $a >> * (b >> a)$

# Hello, Boost.Spirit World!

```
int main(int argc, char* argv[])
{
    char const* in = argv[1];

    if ( phrase_parse( in, in + strlen(in), double_ % ',' , space ) )
    {
        std::cout << "recognized list of floats" << std::endl;
        return EXIT_SUCCESS;
    }
    else return EXIT_FAILURE;
}
```

```
$ test '3.14, 2.72 , 99.9'
```

# Storing Parsers

```
int main(int argc, char* argv[])
{
    char const* in = argv[1];
    rule<char const*> number < double_ % ',';

    if ( phrase_parse( in, in + strlen(in), numbers, space ) )
    {
        std::cout << "recognized list of floats" << std::endl;
        return EXIT_SUCCESS;
    }
    else return EXIT_FAILURE;
}
```

```
$ test '3.14, 2.72 , 99.9'
```

# Storing Parsers

```
int main(int argc, char* argv[])
{
    char const* in = argv[1];
    rule<char const*> numbers;
    numbers = double_ % ',';

    if ( phrase_parse( in, in + strlen(in), numbers, space ) )
    {
        std::cout << "recognized list of floats" << std::endl;
        return EXIT_SUCCESS;
    }
    else return EXIT_FAILURE;
}
```

```
$ test '3.14, 2.72 , 99.9'
```

# Expression Template Gotcha

---

'a' >> 'b'  


\*Note: there are still  
a few things that  
are not parsers

# Expression Template Gotcha

---

```
'a' >> char_('b')
```

A parser, phew!

# Expression Template Gotcha

---

char\_('a') >> 'b'

Also a parser

(At least one operand must be a parser)

# Semantic Actions

```
using namespace phoenix;
```

```
int main(int argc, char* argv[])
{
    double total;
    rule<> number_list =
        double_[ ref(total) = _1 ]
        >> *( ',' >> double_[ ref(total) += _1 ] );
    if ( parse( argv[1], number_list, space ) )
    {
        std::cout << "sum = " << total << std::endl;
        return EXIT_SUCCESS;
    }
    else return EXIT_FAILURE;
}
```

# Static DSEL in C++

---

- Embedded statements are C++ expressions
- Parsed at compile time
- Generates machine-code, executed directly
- Advantages:
  - Syntax-checked by the compiler
  - Better performance (when done right!)
  - Full access to types and data in your program

# Infix Calculator Grammar

## ■ In Extended Backus-Naur Form

```
fact ::= integer | '(' expr ')'  
term ::= fact (('*' fact) | ('/' fact))  
expr ::= term ((+' term) | (-' term))
```

# Infix Calculator Grammar

## ■ In Boost.Spirit

```
using namespace boost::spirit;
typedef qi::rule<std::string::iterator> rule;
rule fact, term, expr;

fact    = int_     | '(' >> expr >> ')';
term    = fact >> *(('*' >> fact) | ('/' >> fact));
expr    = term >> *(('+' >> term) | ('-' >> term));
```

# Balanced, Nested Braces

## ■ In Extended Backus-Naur Form:

```
braces      ::= '{' ( not_brace+ | braces )* '}'  
not_brace  ::= not '{' or '}'
```

## ■ In Spirit.Qi:

```
qi::rule<std::string::iterator> braces, not_brace;  
  
braces      = '{' >> *( +not_brace | braces ) >> '}';  
not_brace  = ~char_("{}");
```

# Spirit.Qi Parser Primitives

Syntax	Meaning
char_	Matches any single character
char_('x')	Match literal character 'x'
char_('a', 'z')	Match characters in the range 'a' through 'z'
char_("1234")	Matches any of '1', '2', '3', or '4'
lit("hello")	Match the literal string "hello"
eo1	Match the end of a line
eoi	Match the end of the input
eps	Matches empty string
eps(p)	Matches empty string; fails if p or p() is false

# Spirit.Qi Parser Operations

Syntax	Meaning
<code>x &gt;&gt; y</code>	Match x followed by y
<code>x   y</code>	Match x or y
<code>~x</code>	Match any char not x (x is a single-char parser)
<code>x - y</code>	Difference: match x but not y
<code>*x</code>	Match x zero or more times
<code>+x</code>	Match x one or more times
<code>-x</code>	x is optional
<code>&amp;x</code>	Positive look-ahead (so-called And-predicate)
<code>!x</code>	Negative look-ahead (so-called Not-predicate)
<code>x[ f ]</code>	Semantic action: invoke f when x matches

# Static DSEL Gotcha!

## ■ Operator overloading woes

```
qi::rule<Iter> r1 = char_ >> 'b'; // OK
qi::rule<Iter> r2 = 'a' >> 'b';      // Oops!
qi::rule<Iter> r3 = + "hello";          // Oops!
```

## ■ At least one operand must be a parser!

```
qi::rule<Iter> r2 = char_('a') >> 'b'; // OK!
qi::rule<Iter> r3 = + lit("hello");      // OK!
```

# Algorithm: `qi::parse()`

```
#include <boost/spirit/include/qi.hpp>

int main()
{
    using namespace boost::spirit;
    qi::rule<char const *> fact, term, expr;

    fact = int_ | '(' >> expr >> ')';
    term = fact >> *(('*' >> fact) | ('/' >> fact));
    expr = term >> *(('+' >> term) | ('-' >> term));

    char const *s1 = "2*(3+4)", *e1 = s1 + std::strlen(s1);
    char const *s2 = "2*(3+4", *e2 = s2 + std::strlen(s2);

    assert(qi::parse(s1, e1, expr) && s1 == e1);
    assert(qi::parse(s2, e2, expr) && *s2 == '*');
}
```

Parse strings as an `expr`  
("start symbol" = `expr`).

`qi::parse` returns a bool and  
mutates begin iterator.

# Algorithm: qi::phrase\_parse()

## ■ Infix Calculator, reloaded

```

using namespace boost::spirit;
qi::rule<char const *, ascii::space_type> fact, term, expr;

fact = int_ | '(' >> expr >> ')';
term = fact >> *((('*' >> fact) | ('/' >> fact));
expr = term >> *((('+' >> term) | ('-' >> term)));

char const *s1 = "2*(3 + 4)", *e1 = s1 + std::strlen(s1);
char const *s2 = "2*(3 + 4", *e2 = s2 + std::strlen(s2);

assert(qi::phrase_parse(s1, e1, expr, ascii::space) && s1 == e1);
assert(qi::phrase_parse(s2, e2, expr, ascii::space) && *s2 == '*');

```

Type of parser used to skip over irrelevant characters.

Instance of that type; eats spaces

# Spirit.Qi Grammars

## ■ Logical grouping of rules

```
template<typename Iter>
struct calc : qi::grammar<Iter, ascii::space_type>
{
    calc() : calc::base_type(expr)
    {
        fact  = int_ | '(' >> expr >> ')';
        term  = fact >> *(('*' >> fact) | ('/' >> fact));
        expr  = term >> *(('+'' >> term) | ('-' >> term));
    }

    qi::rule<Iter, ascii::space_type> fact, term, expr;
};

calc<Iter> c; // our grammar

qi::phrase_parse(iter, end, c, ascii::space);
```

# Semantic Actions

- Action to take when part of your rule succeeds

```
void write(std::vector<char> const & chars)
{
    if(!chars.empty())
        std::cout.write(&chars[0], chars.size());
}
char const *b = "{hi}", *e = b + std::strlen(b);

// This prints "hi" to std::cout
qi::parse(b, e, '{' >> (*ascii::alpha)[&write] >> '}');
```

Match alphabetic characters, call `write()` with the characters that matched.

# Semantic Actions

- Use raw[] to access underlying iterators directly

```
void write(iterator_range<char const *> chars)
{
    std::cout.write(chars.begin(), chars.size());
}

char const *b = "{hi}", *e = b + std::strlen(b);

// This prints "hi" to std::cout
qi::parse(b, e, '{'>> raw[*ascii::alpha][&write] >> '}'');
```

# Semantic Actions

- Some parsers have more useful attributes

```
void write(int d)
{
    std::cout << d;
}
char const *b = "(42)", e = b + std::strlen(b);

// This prints "42" to std::cout
qi::parse(b, e, '(' >> int_&[&write] >> ')');
```

int\_ "returns" an int.

# Semantic Actions

- Some parsers have more useful attributes

```
using lambda::_1;

char const *b = "(42)", e = b + std::strlen(b);

// This prints "42" to std::cout
qi::parse(b, e, '(' >> int_[std::cout << _1] >> ')');
```

We can use Boost.Lambda  
here, too!

# Attributes

- Data associated with a rule invocation.

```
qi::rule<Iter, int(), ascii::space_type> expr;
```

An expr “stack frame” contains an int variable.

When parsing rule expr, its stack frame variable (aka *attribute*) is accessed from actions as \_val.

# Attributes and Phoenix

## ■ Phoenix: the next version of Lambda

```

template <typename Iterator>
struct calculator : grammar<Iterator, int(), space_type>
{
    calculator()
    {
        fact = int_
            | '(' >> expr
            ;
        term = fact
            >> *( '*' >> fact
                | '/' >> fact
                );
        expr = term
            >> *( '+' >> term
                | '-' >> term
                );
    }

    rule<Iterator, int(), space_type> expr, term, fact;
};

rule<Iterator, int(), space_type> fact;
rule<Iterator, int(), space_type> term;
rule<Iterator, int(), space_type> expr;

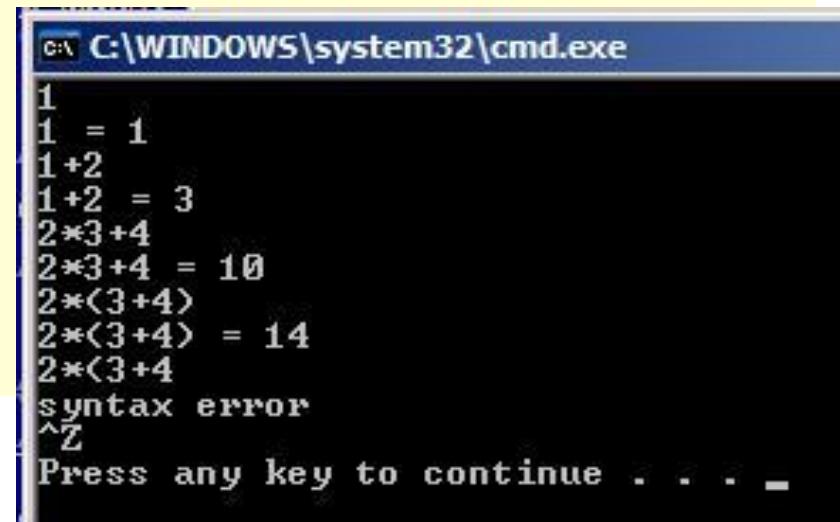
```

# A Calculator that calculates!

```

typedef std::string::const_iterator iterator;
calculator<iterator> def; // Our grammar definition
grammar<calculator<iterator> > calc(def, def.expr); // Our grammar

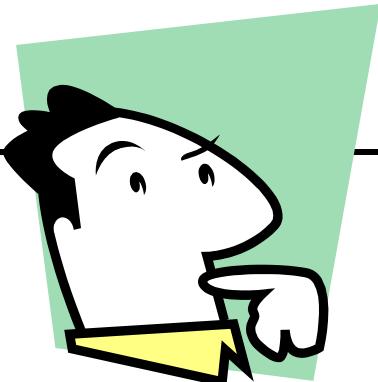
std::string str;
while (std::getline(std::cin, str))
{
    int result = 0;
    iterator iter = str.begin(), end = str.end();
    if(phrase_parse(iter, end, calc, result, space) && iter == end)
    {
        std::cout << str      << " = "
                      << result << '\n';
    }
    else
    {
        std::cout << "syntax error\n";
    }
}
  
```



The screenshot shows a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window displays the output of the calculator program. It processes several mathematical expressions and outputs the results. At the end, it prints 'syntax error' followed by '^Z' and a prompt to press any key to continue.

```

C:\WINDOWS\system32\cmd.exe
1
1 = 1
1+2
1+2 = 3
2*3+4
2*3+4 = 10
2*(3+4)
2*(3+4) = 14
2*(3+4
syntax error
^Z
Press any key to continue . . .
  
```



# Exercise 3

(in progress)

- Use Boost.Spirit to implement a calculator
  - or –
- Use Boost.Spirit to implement a tiny XML parser  
*(students' choice)*

# Should I use Regex or Spirit?

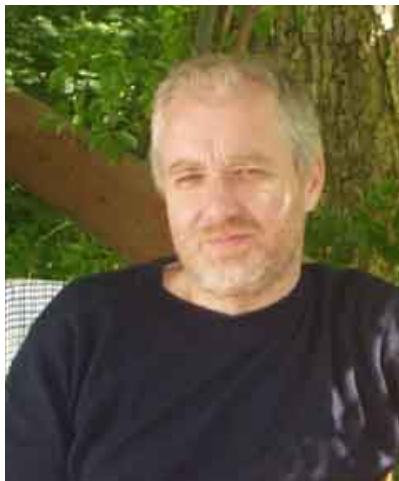
	Regex	Spirit
Ad-hoc pattern matching, regular languages	✓	✓
Structured parsing, context-free grammars	✗	✓
Manipulating text	✓	✓
Semantic actions, manipulating program state	✗	✓
Dynamic; new statements at runtime	✓	✗
Static; no new statements at runtime	✓	✓
Exhaustive backtracking semantics	✓	✗
Blessed by TR1	✓	✗

# Introducing: Spirit.Karma

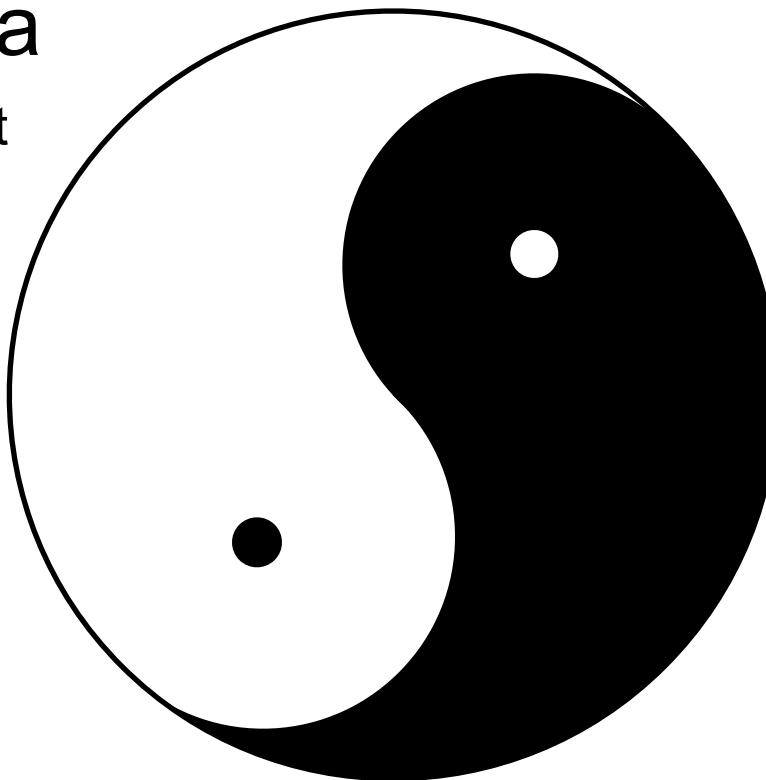
- The yin to Spirit.Qi's yang.

## Spirit.Karma

Structured → Text  
Data



Hartmut Kaiser



## Spirit.Qi

Text → Structured  
Data

# Spirit.Karma

---

- Structured data → Text via Grammars
- Separation of format and data

```

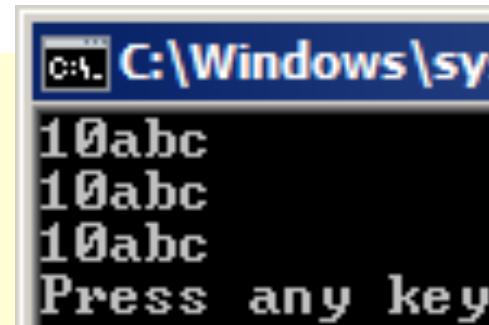
int main()
{
    std::ostream_iterator<char> out(std::cout);

    karma::generate(out, int_(10) << lit("abc"));
    *out++ = '\n';

    karma::generate(out, (int_ << lit)[_1 = val(10), _2 = val("abc")]);
    *out++ = '\n';

    char rg[] = {'a','b','c'};
    std::vector<char> abc(rg, rg+3);
    karma::generate(out, (int_ << *char_)[_1 = val(10), _2 = ref(abc)]);
    *out++ = '\n';
}

```



# Introducing Boost.Xpressive

---

“Why program by hand in five days  
what you can spend five years of your  
life automating?”

-- Terrence Parr, author ANTLR/PCCTS

# Types of DSELs: Dynamic

---

- Example:

```
SQLCommand c = "SELECT * from Employees";
```

- Advantages:

- Unconstrained syntax
  - Statements can be specified at runtime

- Disadvantages:

- Syntax errors discovered at runtime
  - Performance costs of interpretation

# Types of DSELs: Static

---

- Example:

```
double d = (matrix * vector)(3, 4);
```

- Advantages:

- Syntax errors checked at compile-time
  - Aggressive inlining, domain-specific codegen

- Disadvantages:

- Constrained by rules for legal C++ expressions
  - Cannot accept new statements at runtime

# Hrrmmm ...

---



# Boost.Xpressive

- A regex library in the Spirit of Boost.Regex  
(pun intended)
- Both a static and a dynamic DSEL
  - Dynamic syntax is similar to Boost.Regex
  - Static syntax is similar to Boost.Spirit

```
using namespace boost::xpressive;
```

```
sregex dyn = sregex::compile( "Subject: (Re: )?(.*)" );
```

```
sregex sta = "Subject: " >> !(s1= "Re: ") >> (s2= *_);
```

dyn is a dynamic regex

sta is a static regex

# Xpressive Interface

---

- Closely mirrors Boost/TR1 regex:

- `basic_regex<>`
  - `match_results<>`
  - `sub_match<>`
  - `regex_iterator<>`
  - `regex_token_iterator<>`
  - `regex_match()`
  - `regex_search()`
  - `regex_replace()`

- Additions:

- `regex_compiler<>`: a factory for dynamic regex objects
  - Other exciting stuff ...

# Regular Expression

## Expression Template

---

"\\w"	_w
"\\w+"	+_w
"a\\w"	'a' >> _w
"a b"	as_xpr('a')   'b'
"(\\w)\\1"	(s1= _w) >> s1
"[^a-z]"	~range('a', 'z')
"(?=foo)"	before("foo")

# Get a Date

---

```
# Match a date of the form 10-03-2005
```

```
/\d\d?- \d\d?- \d\d(?: \d\d)?/
```

# Get a Date

---

```
// Match a date of the form 10-03-2005
regex date = regex::compile(
    "\\\d\\\\d?-\\\\d\\\\d?-\\\\d\\\\d(?:\\\\d\\\\d)?");
```

# Get a Date

---

```
// Match a date of the form 10-03-2005
regex date = regex::compile(
    "\\\d\\\\d?-\\\\d\\\\d?-\\\\d\\\\d(?:\\\\d\\\\d)?");

regex date =
    _d >> !_d >> '-' // match month
    >> _d >> !_d >> '-' // match day
    >> _d >> _d >> !(_d >> _d); // match year
```

# Regex aliases, anyone?

---

```
regex date = /* ... */;
```

```
// A line in a log file is a date followed by a
// space, and everything up to the newline.
regex log = date >> ' ' >> +~set['\n'];
```

# Semantic Constraints

---

```
// only match valid dates
regex date =
    (_d >> !_d)[if_is_month()] >> '-'
  >> (_d >> !_d)[if_is_day()] >> '-'
  >> (_d >> _d >> !( _d >> _d))
[if_is_year()];
```

# Two great tastes ... ... that taste great together

```
// A line in a log file is a date followed by a
// space, and everything up to the newline.
regex date = regex::compile(get_date_pattern());

regex log = date >> ' ' >> +~set['\n'];
```

# “Impossible” Recursive Regexen!

---

```
regex parens;
parens
    = '('                                // A balanced set of parens ...
      >>                                // is an opening paren ...
      *(                                // followed by ...
        keep( +~(set='(','')') )       // zero or more ...
                                         // of a bunch of things that are
                                         // not parens ...
      |                                // or ...
        by_ref(parens)                 // a balanced set of parens
      )
      >>                                // (ooh, recursion!) ...
      ')'                                // followed by ...
                                         // a closing paren
;
```

# A Regex Calculator?!

---

```
regex group, fact, term, expr;  
  
group = '(' >> by_ref(expr) >> ')';  
fact  = +_d | group;  
term  = fact >> *(('*' >> fact) | ('/' >> fact));  
expr  = term >> *(('+' >> term) | ('-' >> term));
```

# Xpressive 2.0: New Features

---

- Semantic actions
- Custom assertions
- Better errors for invalid static regexes
- Dynamic regex grammars
- Named captures
- Recursive dynamic regexes with (?R) construct
- Range-based regex algorithm interface
- `format_perl`, `format_sed`, and `format_all`
- New: `skip(_s)` modifier, for skipping stuff

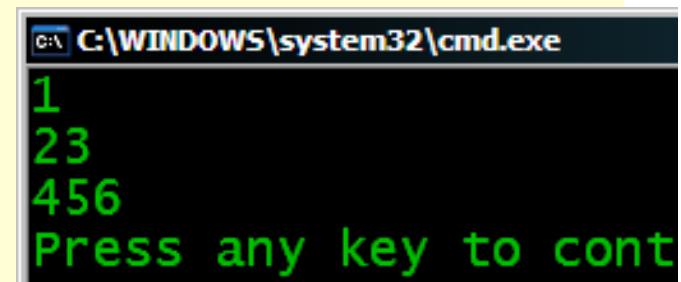
# Xpressive Semantic Actions

## ■ Assign to variables from within a regex

```
// Build a map of strings to integers
std::string str("aaa=>1 bbb=>23 ccc=>456");
std::map<std::string, int> result;

sregex nvpair = ( (s1= +_w) >> ">" >> (s2= +_d) )
    [ ref(result)[s1] = as<int>(s2) ];
sregex rx = nvpair >> *(+_s >> nvpair);

if(regex_match(str, rx))
{
    std::cout
        << result["aaa"] << '\n'
        << result["bbb"] << '\n'
        << result["ccc"] << '\n';
}
```



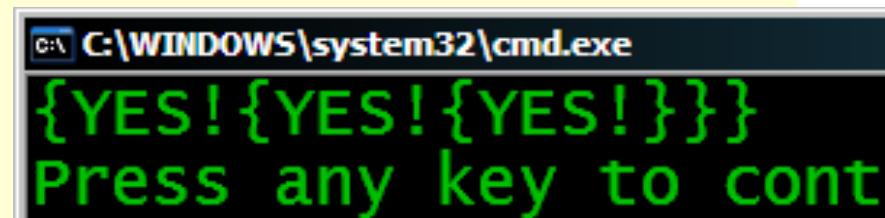
C:\> C:\WINDOWS\system32\cmd.exe  
1  
23  
456  
Press any key to cont

# Dynamic Regex Grammars

- Refer to named regexes from other regexes

```
// Match balanced, nested braces
sregex_compiler comp;
comp.compile("(?$nonbrace=) [^{}]" );
sregex braced = comp.compile(
  "($braced=)\\{((?$nonbrace)+|($braced))*\\}" );

std::string test("{NO! {YES! {YES! {YES! }}}}");
smatch what;
if(regex_search(test, what, braced))
{
    std::cout
        << what[0] << '\n';
}
```



# New: Powerful regex\_replace()

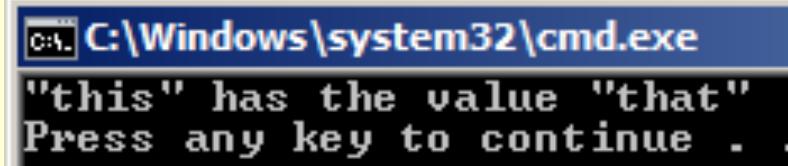
```
#include <map>
#include <iostream>
#include <boost/xpressive/xpressive.hpp>
#include <boost/xpressive/regex_actions.hpp>
using namespace boost::xpressive;

int main()
{
    // Input string
    std::string input("\"$(X)" has the value \"$(Y)\"");

    // Pattern
    sregex envar = "$(" >> (s1= +alnum) >> ")";

    // Environment variables
    std::map<std::string, std::string> replacements;
    replacements["X"] = "this";
    replacements["Y"] = "that";

    std::cout << regex_replace(input, envar, ref(replacements)[s1]) << '\n';
}
```



C:\Windows\system32\cmd.exe  
 "this" has the value "that"  
 Press any key to continue . . .

Lambda for building substitutions

ref(replacements)[s1]

# Boost.Xpressive vs. Boost.Regex

---

- Short Matches:
  - Xpressive always faster than Regex
  - Static Xpressive usually faster than Dynamic
  
- Long Matches (find all matches in complete works of Mark Twain):
  - Regex wins 2/3 of the time

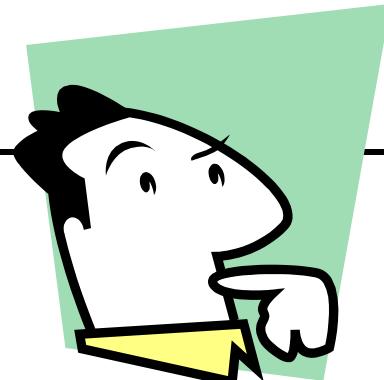
# Xpressive Performance Tips

---

- Use static regexes (worth 10-15%)
- Reuse `match_results<>` objects
- Corollary: Prefer algorithms that take a `match_results<>` object
- Pass iterator ranges, rather than null-terminated strings
- Compile patterns once and reuse
- Use `syntax_option_type::optimize` for often-searched patterns
- Beware nested quantifiers, e.g.  $(a^*)^*$

# Sizing it up

	Regex	Spirit	Xpr
Ad-hoc pattern matching, regular languages	✓	✓	✓
Structured parsing, context-free grammars	✗	✓	✓
Manipulating text	✓	✓	✓
Semantic actions, manipulating program state	✗	✓	✓
Dynamic; new statements at runtime	✓	✗	✓
Static; no new statements at runtime	✓	✓	✓
Exhaustive backtracking semantics	✓	✗	✓
Blessed by TR1	✓	✗	✗



# Exercise 4

## (in progress)

- Use static *and* dynamic xpressive expressions to
  - 1. Replace all occurrences of numbers by the character '#'
  - 2. Find and sum all integer numbers
  - 3. Split the text into tokens delimited by any sequence of punctuation characters or whitespace
  - 4. Print all dates, converting from US format "2/21/2008" to European format "21. 2. 2008"
  - 5. Change all dates from US format "2/21/2008" to European format "21. 2. 2008"

# Part 3: The Secret Stuff

## Hidden support for Unicode

# Wouldn't it be nice ...

---



Hmm ... where,  
oh where, is  
**Boost.Unicode?**

# Unicode Iterators

---

- Implementation detail of Boost.Regex
- Input and Output iterators that convert between Unicode encodings on the fly
- Found in: boost/regex/pending/  
unicode\_iterator.hpp

# Unicode Iterators Example

```
#include <iostream>
#include <iomanip>
#include <boost/regex/pending/unicode_iterator.hpp>

int main()
{
    char const utf8[] = "\xC3\xA4\xC3\xB6\xC3\xBC"; // "äöü"
    boost::u8_to_u32_iterator<char const *>
        begin(utf8), end(utf8 + sizeof(utf8));

    for(; begin != end; ++begin)
    {
        std::wcout << (wchar_t)*begin;
    }
}
```



# Unicode Input Iterators

---

- `u32_to_u8_iterator<>`
  - Adapts sequence of UTF-32 code points to "look like" a sequence of UTF-8.
- `u8_to_u32_iterator<>`
  - Adapts sequence of UTF-8 code points to "look like" a sequence of UTF-32.
- `u32_to_u16_iterator<>`
  - Adapts sequence of UTF-32 code points to "look like" a sequence of UTF-16.
- `u16_to_u32_iterator<>`
  - Adapts sequence of UTF-16 code points to "look like" a sequence of UTF-32.

# Unicode Output Iterators

---

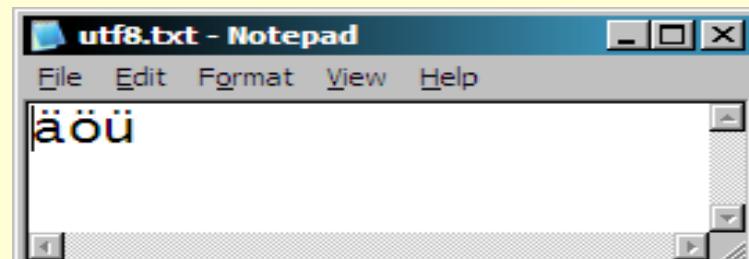
- `utf8_output_iterator<>`
  - Accepts UTF-32 code points and forwards them on as UTF-8 code points.
- `utf16_output_iterator<>`
  - Accepts UTF-32 code points and forwards them on as UTF-16 code points.

# What about IOStreams?

- Can't use the Unicode iterators here ...

```
#include <string>
#include <fstream>

int main()
{
    std::wstring str;
    std::wifstream bad("c:\\utf8.txt");
    bad >> str;
    assert( str == L"äöü" );      // OOPS! :-(
```



# UTF-8 Conversion Facet

---

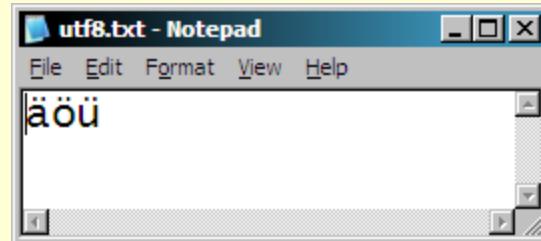
- Converts UTF-8 input to UTF-32
- For use with `std::locale`
- Implementation detail!
- But useful nonetheless



# UTF-8 Conversion Facet

```
#define BOOST_UTF8_BEGIN_NAMESPACE
#define BOOST_UTF8_END_NAMESPACE
#define BOOST_UTF8_DECL
#include <boost/detail/utf8_codecvt_facet.hpp>
#include <libs/detail/utf8_codecvt_facet.cpp>
#include <fstream>

int main()
{
    std::wstring str;
    std::wifstream good("c:\\utf8.txt");
    good.imbue(std::locale(std::locale(),
        new utf8_codecvt_facet));
    good >> str;
    assert( str == L"äöü" );      // SUCCESS!! :-
}
```



# Questions?

