Text Processing With Boost

Or, "How to Beat Perl at Its Own Game"

Talk Overview

Goal: Become adept at C++ string manipulation with the help of Boost.

1. The Simple Stuff
   - Boost.Lexical_cast
   - Boost.String_algo
   - Boost.Tokenizer
   - Boost.Format
2. The Advanced Stuff
   - Boost.Regex
   - Boost.Spirit
   - Boost.Xpressive
3. The Secret Stuff
   - Hidden support for Unicode

Part 1: The Simple Stuff

Utilities for Ad Hoc Text Manipulation

A Legacy of Inadequacy

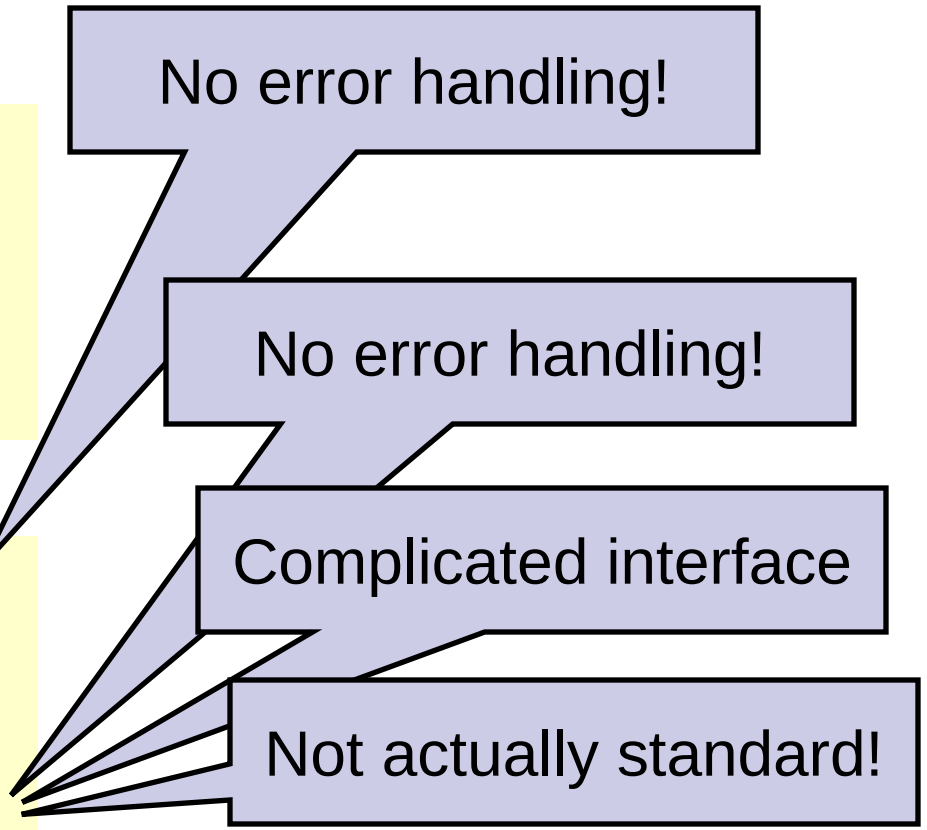
- Python:

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

- C++:

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

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



Stringstream: A Better atoi()

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

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

Boost.Lexical_cast

```
// 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;
}
```

Kevlin Henney

Boost.Lexical_cast

```
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 i18n

Lexical_cast Quiz!

```
// What is i?
int8_t i = lexical_cast<int8_t>( "42" );
```

Hint:

```
typedef char int8_t;
```

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:
>
> ```
>     '   this is a test   '.strip()
> ```
>
> will return the string:
>
> ```
>     'this is a test'
> ```
>
> Lame!
>
> ... There is no standard trim function in C or C++.

Boost.String_algo

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

Pavol Droba

Hello, String_algo!

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

string str1(" hello world! ");
to_upper(str1);  // str1 == " HELLO WORLD! "
trim(str1);      // str1 == "HELLO WORLD!"

string str2 =
    to_lower_copy(
        ireplace_first_copy(
            str1, "hello", "goodbye"));

// str2 == "goodbye world!"
```

Mutate String In-Place

Create a New String

Composable Algorithms!

Plenty of algo's to choose from!

to_upper_copy()
to_upper()
to_lower_copy()
to_lower()
trim_left_copy_if()
trim_left_if()
trim_left_copy()
trim_left()
trim_right_copy_if()
trim_right_if()
trim_right_copy()
trim_right()
trim_copy_if()
trim_if()
trim_copy()
trim()
starts_with()
istarts_with()
ends_with()
iends_with()

ilexicographical_compare()
all()
find_first()
ifind_first()
find_last()
ifind_last()
find_nth()
ifind_nth()
find_head()
find_tail()
find_token()
find_regex()
find()
replace_first()
replace_first_copy()
ireplace_first()
ireplace_first_copy()
erase_first()
erase_first_copy()
ierase_first()

erase_last()
erase_last_copy()
ierase_last()
ierase_last_copy()
replace_nth()
replace_nth_copy()
ireplace_nth()
ireplace_nth_copy()
erase_nth()
erase_nth_copy()
ierase_nth()
ierase_nth_copy()
replace_all()
replace_all_copy()
ireplace_all()
ireplace_all_copy()
erase_all()
erase_all_copy()
ierase_all()
ierase_all_copy()

replace_tail_copy()
erase_tail()
erase_tail_copy()
replace_regex()
replace_regex_copy()
erase_regex()
erase_regex_copy()
replace_all_regex()
replace_all_regex_copy()
erase_all_regex()
erase_all_regex_copy()
find_format()
find_format_copy()
find_format_all()
find_format_all_copy()()
find_all()
ifind_all()
find_all_regex()
split()
split_regex()

contains()
icontains()
equals()
iequals()
lexicographical_compare()
ierase_first_copy()
replace_last()
replace_last_copy()
ireplace_last()
ireplace_last_copy()
replace_head()
replace_head_copy()
erase_head()
erase_head_copy()
replace_tail()
join
join_if()

String_algo: `split()`

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

split( tokens, str, 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 iterface, *or*
  - Iterator interface
  - Parsing done lazily

John R. Bandela

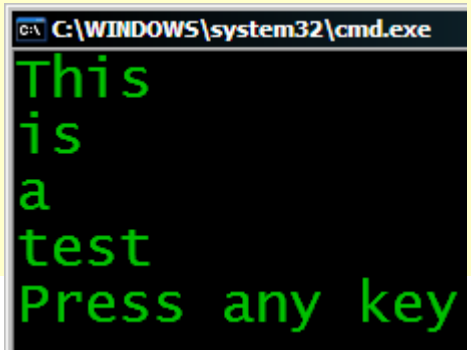
Hello, Tokenizer!

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

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

tokens behaves like a container of tokens

*beg returns a reference to an internal std::string

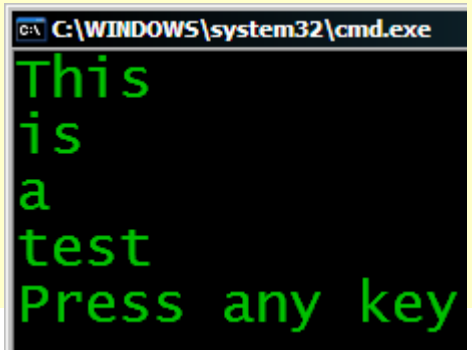


char_separator<>

```
#include <string>
#include <iostream>
#include <boost/foreach.hpp>
#include <boost/tokenizer.hpp>
using namespace boost;

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

Specify the separator characters



Other Tokenizer Functions

- `escaped_list_separator<>`
  - Useful for tokenizing comma-separated lists, where *escaped* commas are *not* delimiters.
- `offset_separator`
  - Breaks strings at integer offsets
- Others, defined by you!
  - 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: "i = ***    0xffff***"
std::printf("i = ***%#10x***\n", i);

// The equivalent using C++ IOStreams ...
std::cout << "i = ***"
          << std::setiosflags(std::ios_base::showbase)
          << std::setw(10)
          << std::hex
          << i
          << std::resetiosflags(std::ios_base::showbase)
          << "***\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: "i = ***    0xffff***"
std::printf("i = ***%#10x***\n", i);

// The equivalent using Boost.Format ...
std::cout << boost::format("i = ***%#10x***\n") % i ;
```

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

Hello, Boost.Format

- Positional arguments:

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

- `printf()`-style formats:

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

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

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

- Formatting can be used with arguments, too:

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

Text Formatting Grudge Match

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

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: "))
      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 += ...;
}
```

- ✘ Describes algorithm
- ✘ Verbose
- ✘ Hard to maintain

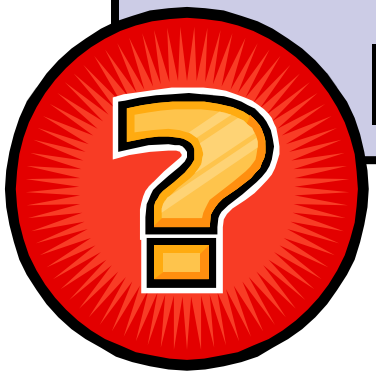
Declarative:

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

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

Riddle me this ...

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

Exercise: 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. "Bold!"

```
"\\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');
```

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
   braces("{[^{}]*({[^{}]*({[^{}]*({[^{}]*({[
    ({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*
    ({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*
    ({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*
    ({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*
    ({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*
    ({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*
    ({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*
    ({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*
    ({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*
    ({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*({[^{}]*
```

Whoops!

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 Boost.Spirit

- 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

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

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

Infix Calculator Grammar

- In Boost.Spirit

```
spirit::rule<> group, fact, term, expr;

group   = '(' >> expr >> ')';
fact    = spirit::int_p | group;
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 Boost.Spirit:

```
rule<> braces, not_brace;

braces      = '{' >> *( +not_brace | braces ) >> '}';
not_brace   = ~chset_p("{}");
```

Spirit Parser Primitives

Syntax	Meaning
`ch_p('X')`	Match literal character 'X'
`range_p('a','z')`	Match characters in the range 'a' through 'z'
`str_p("hello")`	Match the literal string "hello"
`chseq_p("ABCD")`	Like str_p, but ignores whitespace
`anychar_p`	Matches any single character
`chset_p("1234")`	Matches any of '1', '2', '3', or '4'
`eol_p`	Matches end-of-line (CR/LF and combinations)
`end_p`	Matches end of input
`nothing_p`	Matches nothing, always fails

Spirit Parser Operations

Syntax	Meaning
`x >> y`	Match x followed by y
`x | y`	Match x or y
`~x`	Match any char not x (x is a single-char parser)
`x - y`	Difference: match x but not y
`*x`	Match x zero or more times
`+x`	Match x one or more times
`!x`	x is optional
`x[ f ]`	Semantic action: invoke f when x matches

Static DSEL Gotcha!

- Operator overloading woes

```
rule<> r1 = anychar_p >> 'b';  // OK
rule<> r2 = 'a' >> 'b';        // Oops!
rule<> r3 = + "hello";         // Oops!
```

- At least one operand must be a parser!

```
rule<> r2 = ch_p('a') >> 'b';  // OK!
rule<> r3 = + str_p("hello");  // OK!
```

Storing rules

- `spirit::rule<>`
  - polymorphic rule holder

```
spirit::rule<> not_brace = ~chset_p("{}");
```

- Caveat: `rule<>` can be tricky
  - Assignment operator has unusual semantics.
  - Cannot put them in std containers!
- `stored_rule<>` for value semantics

```
spirit::stored_rule<> storable = not_brace.copy();
```

Algorithm: spirit::parse()

```
#include <boost/spirit.hpp>
using namespace boost;

int main() {
    spirit::rule<> group, fact, term, expr;

    group   = '(' >> expr >> ')';
    fact    = spirit::int_p   | group;
    term    = fact >> *(('*' >> fact) | ('/' >> fact));
    expr    = term >> *(('+' >> term) | ('-' >> term));

    assert( spirit::parse("2*(3+4)", expr).full );
    assert( ! spirit::parse("2*(3+4", expr).full );
}
```

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

spirit::parse returns a spirit::parse_info<> struct.

Algorithm: `spirit::parse()`

- Infix Calculator, reloaded

```
spirit::rule<spirit::phrase_scanner_t> group, fact,
   term, expr;
// ...

spirit::parse_info<> info =
   spirit::parse("1 + 3", expr, spirit::space_p);
if(info.full)
{
    std::cout << "success!" << std::endl;
}
```

Instance of that type; eats spaces

Type of parser used to skip over irrelevant characters.

Spirit Grammars

- Group rules together
- Decouple "tokenizing/lexing/scanning"

```
struct calculator : spirit::grammar<calculator>
{
  template<class Skip> struct definition
  {
    spirit::rule<Skip> group, fact, term, expr;
    definition(calculator const & self)
    {
      // ...
      expr = term >> *(('+' >> term) | ('-' >> term));
    }
    rule<Skip> const & start() const { return expr; }
  };
};
```

Algorithm: spirit::parse()

- Infix Calculator, revolutions

```
struct calculator : spirit::grammar<calculator> {
  // ...
};

// This works!
spirit::parse("1+3", calculator());

// This works, too!
spirit::parse("1 +3", calculator(), spirit::space_p);
```

Semantic Actions

- Action to take when part of your grammar succeeds

```
void write(char const *begin, char const *end)
{
  std::cout.write(begin, end - begin);
}

// This prints "hi" to std::cout
spirit::parse("{hi}", '{' >> (*alpha_p)[&write] >> '}');
```

Match alphabetic characters, call write() with range of characters that matched.

Semantic Actions

- A few parsers process input first

```
using namespace lambda;

// This prints "42" to std::cout
spirit::parse("(42)", '(' >> int_p[cout << _1] >> ')');
```

We can use a Boost.Lambda expression as a semantic action!

Closures

- Data associated with a rule.

```
struct calc_closure
  : spirit::closure<calc_closure, double>
{
    member1 val;
};

rule<Scan, calc_closure::context_t> expr;
```

A `calc_closure` "stack frame" contains a double named `val`.

When parsing rule `expr`, its `calc_closure` stack frame variable is accessed as `expr.val`.

Closures and Phoenix

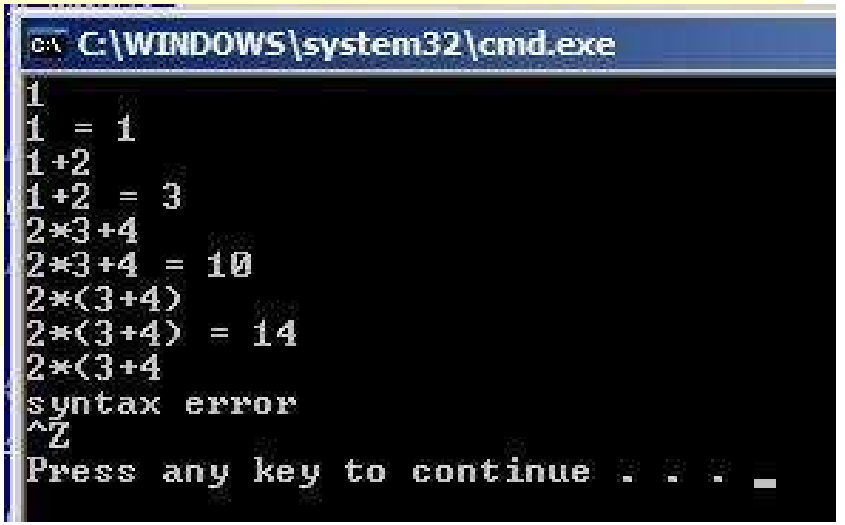
- Phoenix: the next version of Lambda

```
struct calculator : spirit::grammar<calculator, calc_closure::context_t>
{
  template<class Scan> struct definition
  {
    spirit::rule<Scan, calc_closure::context_t> group, fact, term, expr;
    spirit::rule<Scan> top;
    definition(calculator const & self)
    {
      using namespace phoenix;
      top     = expr[ self.val = arg1 ];
      // ...
      expr    = term[ expr.val = arg1 ]
                  >> *(('+' >> term[ expr.val += arg1 ])
                     | ('-' >> term[ expr.val -= arg1 ]));
    }
    spirit::rule<Scan> const & start() const { return top; }
  };
};
```

A Calculator that calculates!

```
using namespace phoenix;
calculator calc;
std::string str;

while (std::getline(std::cin, str))
{
  double n = 0;
  if (spirit::parse(str.c_str(), calc[ var(n) = arg1 ]).full)
  {
    std::cout << str << " = " << n << std::endl;
  }
  else
  {
    std::cout << "syntax error\n";
  }
}
```



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	✓	✗

A Peek at 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 ...

Xpressive: A Mixed-Mode DSEL

- Mix-n-match static and dynamic regex

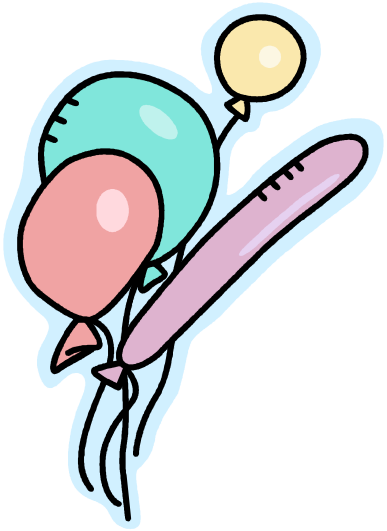
```
// Get a pattern from the user at runtime:
std::string str = get_pattern();
sregex pat = sregex::compile( str );

// Wrap the regex in begin- and end-word assertions:
pat = bow >> pat >> eow;
```

- Embed regexes by reference, too!

```
sregex braces, not_brace;

not_brace = ~(set= '{', '}');
braces = '{' >> *(+not_brace | by_ref(braces)) >> '}';
```



Announcing ...

Boost.Xpressive 2.0,
available *now* in CVS!

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`

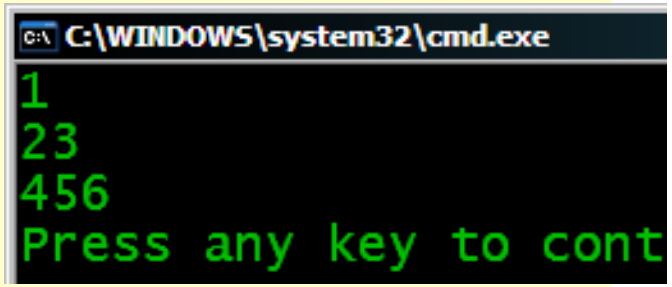
New! Xpressive Semantic Actions

- 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';
}
```

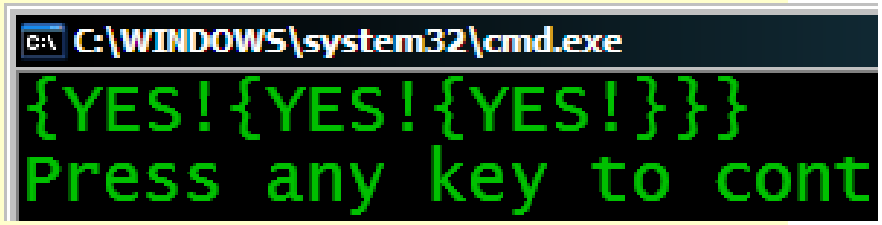


New! Dynamic Regex Grammars

- 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';
}
```



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	✓	✗	✗

Part 3: The Secret Stuff

Hidden support for Unicode

Wouldn't it be nice ...

Unicode Iterators

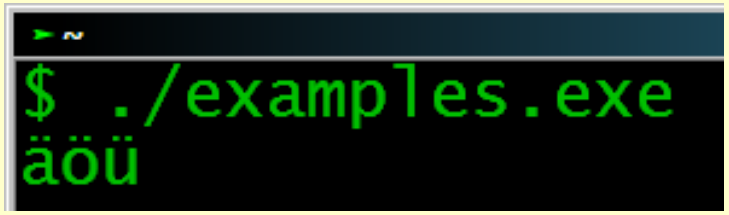
- Implementation detail of Boost.Regex
- Input and Output iterators that convert between Unicode encodings on the fly
- Found in:
  `boost/regex/pending/unicode_iterator.hpp`

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

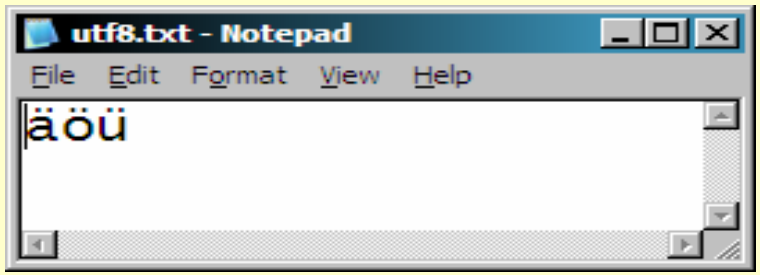
- `utf8_output_iterator<>`
  - Accepts UTF-32 code points and forwards them on as UTF-8 code points.
- `utf16_output_iterator<>`
  - Accepts UTF-32 code points and forwards them on as UTF-16 code points.

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! :-(
}
```



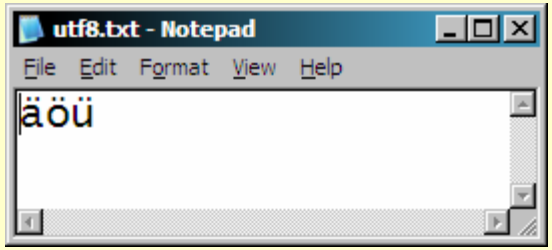
UTF-8 Conversion Facet

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



Questions?