Boost.Wave

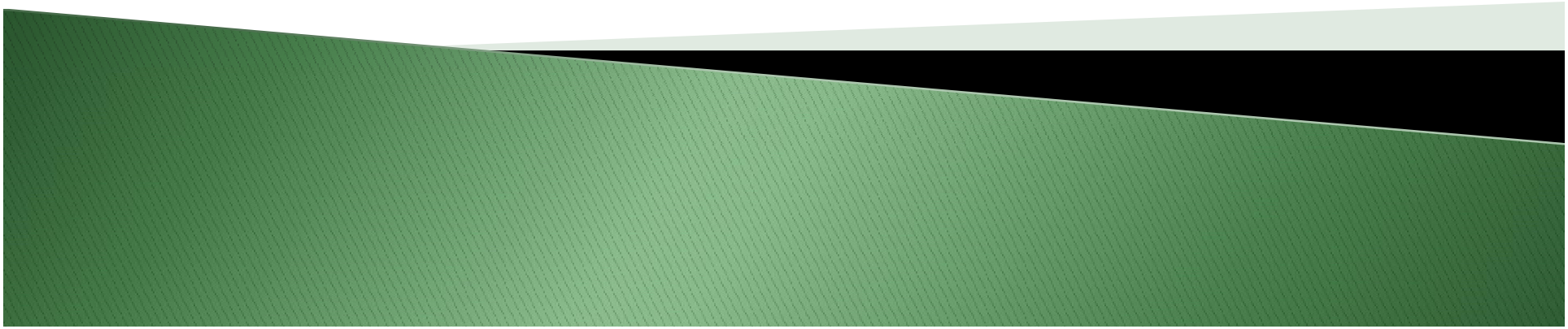
A Standards conformant C99/C++ preprocessor library

Hartmut Kaiser

hartmut.kaiser@gmail.com

Center for Computation and Technology

Louisiana State University



Agenda

- Overview
  - Some History
- Quickstart
- Library Structure
- Library Features
- The Context Object
  - Include search paths
  - Macro Definitions
  - Language Options
  - Supported `#pragma` Directives
- Build Configuration
- Examples

Overview

Some History

- Started in 2001 as an experiment in how far Spirit could be taken
  - Initially it was not meant to be high performance
  - Merely a experimental platform
  - Almost no Standards conformant preprocessor available
- First full rewrite in 2003
  - Implemented macro namespaces which lead to some discussions in the C++ committee (removed now)
  - Macro expansion trace, which is still a unique feature
- In Boost since 2005 (V1.33.0)
  - At this time one of the 2 available fully conformant preprocessors (gcc is the other)

Some History

- Got a comprehensive regression test suite with over 400 single unit tests
- Continuous improvement and development
  - Based on user feedback
- A lot of improvements since then
  - Performance now on par with commercial preprocessors (but still slower than gcc or clang)
  - Usability improvements helping to use Wave as a library

Overview

- Wave is a Standards conformant implementation of the mandated C99/C++98 preprocessor functionality
  - C++0x has been aligned with C99
    - Variadic macros, placeholders, operator _Pragma()
  - Missing C++0x features are under development
    - Unicode: u"UTF-16", U"UTF-32", u8"UTF-8", new language features etc.
- Generates sequence of C99/C++ tokens exposed by an `iterator` interface
- It's a library which is easy to use and configure
  - But allows to fine grained tweaking of behavior
  - Simple expansion of functionality by user supplied code

Overview: What's New in V2

- Streamlined and unified API
- New macro expansion engine
  - Work in progress
- New documentation (QuickBook based)
- Started to move to Spirit V2
- Performance enhancements
- Enhanced and extended test suite

- Incompatible to earlier versions, will work with Boost V1.36 and above only

Quickstart

Quickstart

```
//  The template boost::wave::cpplexer::lex_token<> is the token type
    typedef boost::wave::cpplexer::lex_token<> token_type;

//  The template boost::wave::cpplexer::lex_iterator<> is the lexer type
    typedef boost::wave::cpplexer::lex_iterator<token_type> lex_iterator_type;

//  This is the resulting context type to use.
    typedef boost::wave::context<std::string::iterator, lex_iterator_type>
        context_type;

// The preprocessing of the input stream is done on the fly behind the
// scenes during iteration over the context_type::iterator_type stream.
    context_type ctx (instring.begin(), instring.end(), argv[1]);

// At this point you might want to set the parameters for the preprocessing
// such as include paths, predefined macros, other options
    ctx.add_include_path("...");

// analyze the input file
    context_type::iterator_type first = ctx.begin();
    context_type::iterator_type last = ctx.end();

    while (first != last) {
        std::cout << (*first).get_value();
        ++first;
    }
```

Quickstart

- The context object takes a pair of iterators and a filename.
  - Iterators supply data to be processed
  - Filename indicates current start context (file)

```
context_type ctx (instring.begin(), instring.end(), argv[1]);
```

- Change parameters and options
  - Include paths

```
ctx.add_include_path("...");
```

  - (Pre-)define macros

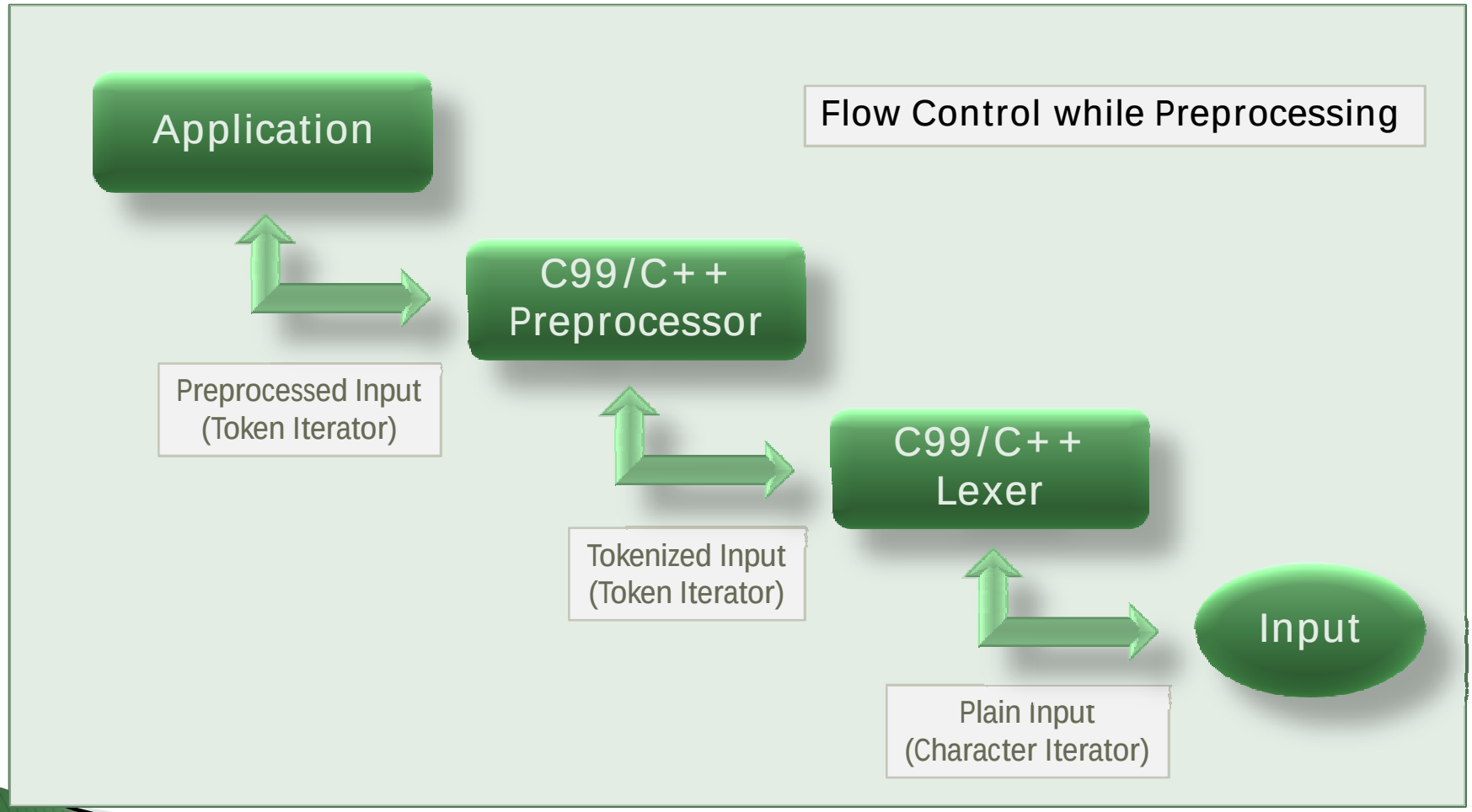
```
ctx.add_macro_definition(...);
```

  - Set options

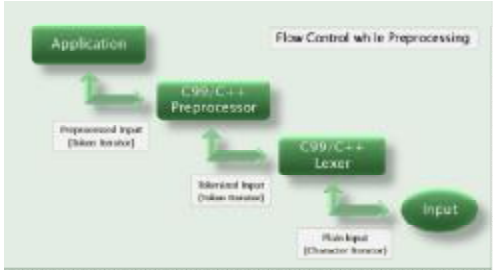
```
ctx.set_language(...);
```

Library Structure and Features

Library Structure

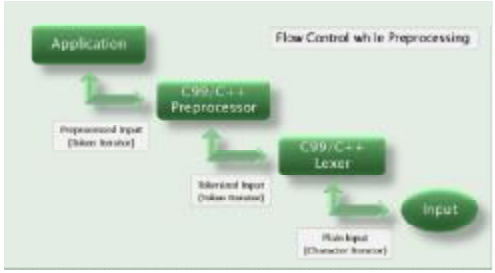


Library Features



- C99/C++ tokenization
  - C++ tokens with position in input (filename, line, column)
  - Usable as the input for parsing (plays well with Spirit)
  - Lexer components are usable standalone
- Preprocessing
  - File inclusion (`#include <>`, `#include ""`, ...)
  - Conditional compilation (`#if`/`#else`, `#ifdef`, ...)
  - Macro definition and expansion (`#define`, `#undef`)
  - Line control (`#line` handling)
  - Specific options (`#pragma` handling)
- Supports all C99 features (optionally even in C++)
  - Variadic macros (variable number of macro arguments)
  - Placeholders (empty macro arguments)
  - Well defined token concatenation
  - Support of `operator _Pragma()`

Library Features



- Include guard detection for automatic `#pragma once` functionality (heuristic)
- Sophisticated whitespace handling
  - Optional whitespace reduction/preservation in output
  - Whitespace insertion for automatic output disambiguation
- Preprocessing hooks
  - Functions invoked at certain points during the processing of the input
    - On macro replacement, file inclusion, evaluation of compilation conditions, custom `#pragma` directives, whitespace handling, etc.
- Serialization
  - Store internal state of the preprocessing
    - Macros, include paths, language options, options selected at compile time, etc.

The Context Object

The Context Object

- `boost::wave::context`
  - Is the main user visible object of the Wave library
  - Set options for preprocessing
    - Set include paths, (pre-)define and undefine macros, set language and preprocessing options
  - Returns pair of iterators providing the generated tokens while de-referenced
  - Provides macro introspection functionality for defined macros
    - Iteration over macro names
    - Introspect macro definitions (type, parameter names, replacement text)
  - May work in interactive mode
  - Serialization support (`boost::serialization`)

The Context Object

- Template parameters
  - `Iterator`: type of underlying input sequence
  - `Lexer`: type of lexer object to use
    - Wave contains 3 predefined C99/C++ lexers (Re2C, SLex, Boost.Lexer)
  - `[InputPolicy]`: customize the way, how an included file is to be represented by a pair of iterators
  - `[Hooks]`: define functions to be called by the library in different contexts
  - `[Derived]`: enable the derivation of a customized context object

The Context Object

- Constructor

```
context(target_iterator_type const& first,
        target_iterator_type const& last,
        char const* filename = "<Unknown>",
        HooksT const& hooks = HooksT());
```

- Get main iterators

```
begin(), end()
begin(target_iterator_type const& first,
      target_iterator_type const& last);
```

Include Search Paths

- Wave maintains two separate search paths for files to include
  - System search path: `#include <…>`
  - Normal search path: `#include "…"`

```
ctx.add_include_path("…");
ctx.add_sys_include_path("…");
```

- Allows to work in two modes
  - With or without implicit current directory lookup for `#include "…"`
    - [Current directory **à**]
      Normal search path **à** System search path
- Support for `#include_next` (optional)
  - Looks for the next matching file in the search paths

Macro Definitions

- Return pair of iterators allowing to enumerate defined macros

```
ctx.macro_names_begin()
ctx.macro_names_end()
```

- Macro introspection

```
ctx.get_macro_definition()
ctx.is_defined_macro()
```

- Define and undefine macros

```
ctx.add_macro_definition()
ctx.remove_macro_definition()
```

- Possible to use command line syntax
  - "MACRO(x)=definition"

Language Options

- Main modes C99/C++ (soon FORTRAN)
  - Mainly different token types (i.e. "->*" is not a C token)
  - C99 macro features (variadics et.al.)
- Options
  - Support long long suffixes (i.e. `0LL`, `0i64`)
  - Support variadics (et.al.) in C++
  - Insert whitespace (for disambiguation)
  - Preserve whitespace and/or comments (*)
  - Convert trigraphs (i.e. "??/" **à** "\\")
  - Emit `#line` directives (*)
  - Include guard detection
  - Emit (pass through) non-recognized `#pragma` directives

Supported #pragma Directives

- `#pragma once`
  - Avoid opening included files more than once
  - Automatic include guard detection (optional)
- `#pragma message("")`
  - Allow to print messages during preprocessing
- `#pragma wave command[(options)]`
  - Custom (application defined) `#pragma` implementations using preprocessing hooks
    - Wave tool:
      - `#pragma wave trace(on/off)`
      - `#pragma wave timer(restart/resume/off)`
      - `#pragma wave system(command)`
      - `#pragma wave stop("")`
      - `#pragma wave option(line: on/off/push/pop)`

The Token Interface

- boost::wave::cpplexer::lex_token< >
  - Default implementation provided by the library
  - It is possible to use your own token type
  - Needs to expose a simple interface

```
contructor(token_id id, string_type const& value,
  position_type const& pos);

operator token_id() const;
string_type const& get_value() const;
position_type const& get_position() const;
void set_token_id (token_id id);
void set_value (string_type const& newval);
void set_position (position_type const& pos);
```

Error Handling

- Errors and Warnings during preprocessing are reported as special exceptions:

```
wave::preprocess_exception
wave::macro_handling_exception
wave::cpplexer_exception
```

- Usually possible to continue after exception
  - Code restores valid state allowing to continue
  - Exceptions ‘know’ if it’s safe to continue

```
bool wave::is_recoverable(e);
```

Build Configuration

Compile Time Configuration

Compile Time Constant	Description	Default
SUPPORT_WARNING_DIRECTIVE	Support #warning directive	1
SUPPORT_MS_EXTENSIONS	Support #region, #endregion, __int64, etc.	Platform
PREPROCESS_ERROR_MESSAGE_BODY	Preprocess body of #error directive	1
EMIT_PRAGMA_DIRECTIVES	Emit unknown #pragma directives (after preprocessing)	1
PREPROCESS_PRAGMA_BODY	Preprocess body of #pragma directive	1
ENABLE_COMMANDLINE_MACROS	Support syntax: '-DMACRO(x,y)=x+y'	1
STRINGTYPE	Use custom type compatible to std::string	flex_string
SUPPORT_VARIADICS_PLACEMARKERS	Support variadics, placeholders and well defined token concatenation	1
MAX_INCLUDE_LEVEL_DEPTH	Set maximum include level depth	1024
SUPPORT_PRAGMA_ONCE	Support #pragma once	1
SUPPORT_PRAGMA_MESSAGE	Support #pragma message("…")	1
SUPPORT_INCLUDE_NEXT	Support #include_next	1
USE_STRICT_LEXER	Identifier names may not contain '$'	0
PRAGMA_KEYWORD	Define keyword for #pragma command extensions	"wave"
SERIALIZATION	Support serialization (using Boost.Serialization)	0
SUPPORT_THREADING	Support threading support	Build type
SUPPORT_LONGLONG_INTEGER_LITERALS	Internal integer representation is long long	0

Separation Compilation Model

- Objects in Wave are large templates
  - Including everything into one compilation unit leads to long compilation times
- Wave uses separation compilation model
  - Main compilation unit (the one using Wave) includes template declarations only
  - Additional files include template definitions and use explicit template specialization
- Advantage
  - Better compilation times
- Disadvantage
  - Explicit specialization needs to ‘know’ iterator type used for instantiation of lexer and token types

Separation Compilation Model

- Wave library contains by default
  - Lexer specializations for `std::string::iterator` and `std::string::const_iterator`
  - Preprocessor related template specializations for default token type (`boost::wave::cpplexer::lex_token<>`)
- Add your own specializations
  - If different token type needs to be used
  - If another iterator type for the underlying input stream needs to be used
- See examples (cpp_tokens, list_includes, real_positions, waveidl)

Examples

Examples

- Quick_start
  - As simple as it can get
- Lexed_tokens
  - Use of the predefined lexer (Re2C) without using the preprocessor
- List_includes
  - List full paths of included files
  - Use different lexer (Boost.Lexer) with predefined token type
- Cpp_tokens
  - Print verbose information about generated tokens
  - Use different lexer (SLex) with custom token type
- Advanced_hooks
  - Output code from conditional code sections normally not evaluated (but commented)
- Preprocess_pragma_output
  - Insert preprocessed code sequences using a custom `#pragma`
- Real_positions
  - Correct positions in returned tokens to appear consecutive
  - Use custom token type
- Hannibal
  - Partial C++ parser using Spirit
- Waveidl
  - IDL (interface definition language) oriented preprocessor
  - Use a custom lexer with predefined token type

Known Applications

- Synopsis (http://synopsis.fresco.org/)
  - A Source-code Introspection Tool
- ROSE (http://rosecompiler.org/)
  - A Tool for Building Source-to-Source Translators
- Vera++ (http://www.inspirel.com/vera/)
  - Programmable verification and analysis tool for C++
- Zoltán Porkoláb
  - Debugger for C++ Templates
- Hannibal
  - Partial C++ parser using Spirit
- Wave tool (http://www.boost.org/doc/tools.html)
  - Full blown preprocessor

- If you know of more, please drop me a note