Design: clang-format

This document contains a design proposal for a clang-format tool, which allows C++ developers to automatically format their code independently of their development environment.

Context

While many other languages have auto-formatters available, C++ is still lacking a tool that fits the needs of the majority of C++ programmers. Note that when we talk about **formatting** as part of this document, we mean both the problem of **indentation** (which has been largely solved independently by regexp-based implementations in editors / IDEs) and **line breaking**, which proves to be a harder problem.

There are multiple challenges to formatting C++ code:

- a vast number of different coding styles has evolved over time
- many projects value consistency over conformance and dislike style-only changes, thus making it important to be able to work with code that is not written according to the most current style guide
- macros need to be handled properly
- it should be possible to format code that is not yet syntactically correct

Goals

- Format a whole file according to a configuration
- Format a part of a file according to a configuration
- Format a part of a file while being consistent as best as possible with the rest of the file, while falling back to a configuration for options that cannot be deduced from the current file
- Integrating with editors so that you can just type away until you're far past the column limit, and then hit a key and have the editor layout the code for you, including placing the right line breaks
- Being usable from C++ tools to fix up formatting of code that has been changed due to a refactoring step
- Develop as a library and Integrate into libclang

Non-goals

- Identifying / fixing a full style guide and static analysis; we want a different tool for changes that potentially change semantics; clang-format will be a precondition for such a tool, but has much narrower scope
- Indenting code while you type; this is a much simpler problem, but has even stronger performance requirements - the current editors should be good enough, and we'll allow

new workflows that don't ever require the user to break lines
- The only lexical elements clang-format should touch are: whitespaces, string-literals and comments. Any other changes ranging from ordering includes to removing superfluous parentheses are not in the scope of this tool.
- Per-file configuration: be able to annotate a file with a style which it adheres to (?)

Code location

Clang-format is a very basic tool, so it might warrant living in clang mainstream. On the other hand it would also fit nicely with other clang refactoring tools. **TODO: Where do we want clang-format to live?**

Parsing approach

The key consideration is whether clang-format can be based **purely on a lexer**, or whether it needs type information, and we need the full **AST**. We considered how far we would get with a lexer based approach and found that even for basic indenting type information is sometimes necessary, and a lot of layout decisions depend on the type, too. Thus, we do not think it makes sense to have a lexer based approach - one could imagine a "clang-quick-indent" tool that is purely lexer based, but that would need to be a different tool. We might want to spawn such a tool out of clang-format, since we'll need to handle cases that are not represented in the AST, but that is not a priority.

Examples:

AST-dependent indentation:

```
callFunction(foo<something,
                  ^ line up here, if foo is a template name
             ^ line up here otherwise
```

AST-dependent line breaking:
Detecting that '*' is an binary operator in this case requires parsing; if it is a binary operator, we want to line-break after it, if it is a unary operator, we want to prevent line breaking

```
result = variable1 * variable2;
```

AST-dependent whitespace inside lines:

```
a * b;
   ^ Binary operator or pointer declaration?
a & f();
   ^ Binary operator or function declaration?
```

Challenge: Preprocessor

Not every line in a program is covered by the AST - for example, there are unused macro definitions, various preprocessor directives, #ifdef'ed out code, etc.

We will at least need some form of lexing approach for the parts of a source file that cannot be correctly indented / line broken by looking at the AST.

Algorithm

Visit all nodes on the AST; for each node that is part of a macro expansion, consider all locations taking part in that macro expansion. If the location is within the range that need to be indented, look at the code at the location, the rules around the node, and adjust whitespace as necessary. If the node starts a line, adjust the indent; if a node overflows the line, break the line.
TODO: figure out what to do with the lines that are not visited that way.

Configuration

To support a majority of developers, being able to configure the desired style is key. We propose using a YAML configuration file, as there's already a YAML parser readily available in LLVM. Proposals for more specific ideas welcome.

Style deduction

When changing the format of code that does not conform to a given style configuration, we will optionally try to deduce style options from the file first, and fall back to the configured layout when there was no clear style deducible from the context.
TODO: Detailed design ideas.

Interface

This is a strawman. Please shoot down.

Command line interface:
Command line interfaces allow easy integration with existing tools and editors.

USAGE: clang-format <build-path> <source> [<range0> [<range1> ...]] [-- list of command line arguments to the parser]

<rangeN>: Specifies a code range to be reformatted; if no code range is given, assume the whole file. The format of the range is "<start_line>:<start_column>-<end_line>:<end_column>".

Code level interface:
Reformatting source code is also a prerequisite for automated refactoring tools. We want to be able to integrate the reformatting as a post-processing step on top of other code transformations to make sure as little human intervention is needed as possible.

How this is designed will highly depend on the level of parsing necessary.

Competition

TODO: List other formatting tools we're aware of and how well they work

- GNU indent - C only;
- BCPP (http://invisible-island.net/bcpp/bcpp.html) - "it does (by design) not attempt to wrap long statements"; written in about 1995, since then had very few changes;
- Artistic Style (http://astyle.sourceforge.net/) - one of the most frequently used, but "not perfect";
- Uncrustify (http://uncrustify.sourceforge.net/) - has lots of configuration options;
- GreatCode (http://sourceforge.net/projects/gcgreatcode/) - not supported since 2005;
- Style Revisor (http://style-revisor.com) - commercial; claims to understand C++, but it isn't released yet, so no way to try; uses code snippets to specify rules.

All of them except Style Revisitor seem to have simplistic regexp-based c++ parsing.