UB? In My Lexer?

Abstract

The mere act of Lexing c++ can result in undefined behavior. This paper removes that undefined behavior. Further work will be needed to remove all undefined behavior in [cpp].

Motivation

According to the standard, the following examples expose undefined behavior:

```
int \ // UB : universal character name across spliced lines
u\n0\n3\n9\n1 = 0;
```

```
#define CONCAT(x, y) x ## y
int CONCAT(\, u0393) = 0; // UB: universal character name formed by macro expansion
```

```
// UB: unterminated string
const char * foo = "
```

It does seem unfortunate that lexing C++ would incur UB. As such, we propose to change the specification to remove the UB by either well-defining the behavior or making it ill-formed, closely matching implementations. The status-quo, as well as the proposed changes are summarized below. The red cell highlights the only impact this paper would have on exiting implementations.

<table>
<thead>
<tr>
<th></th>
<th>GCC</th>
<th>CLANG</th>
<th>EDG</th>
<th>MSVC</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spliced UCN</td>
<td>Supported</td>
<td>Supported</td>
<td>Supported</td>
<td>Error</td>
<td>Well-formed</td>
</tr>
<tr>
<td>UCN Produced BY ##</td>
<td>Supported</td>
<td>Supported</td>
<td>Supported</td>
<td>Supported</td>
<td>Well-formed</td>
</tr>
<tr>
<td>Unterminated “ ot’</td>
<td>ill-formed</td>
<td>ill-formed</td>
<td>ill-formed</td>
<td>ill-formed</td>
<td>ill-formed</td>
</tr>
</tbody>
</table>
We propose that spliced UCNs be supported because, in addition of 3/4 of surveyed compilers supporting it, it falls off naturally of the specification: splicing happens before any other form of tokenization, and supporting it avoid special casing this oddity.

Wording

**Phases of translation**

2. Each sequence of a backslash character (\) immediately followed by zero or more whitespace characters other than new-line followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. **Except for splices reverted in a raw string literal, if a splice results in a character sequence that matches the syntax of a universal-character-name, the behavior is undefined.** A source file that is not empty and that does not end in a new-line character, or that ends in a splice, shall be processed as if an additional new-line character were appended to the file.

A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. In this document, glyphs are used to identify elements of the basic character set. The categories of preprocessing token are: header names, placeholder tokens produced by preprocessing `import` and `module` directives (`import-keyword`, `module-keyword`, and `export-keyword`), identifiers, preprocessing numbers, character literals (including user-defined character literals), string literals (including user-defined string literals), preprocessing operators and punctuators, and single non-whitespace characters that do not lexically match the other preprocessing token categories. If a U+0027 APOSTROPHE or a U+0022 QUOTATION MARK matches the last category, the behavior is undefined. A source file that is not empty and that does not end in a new-line character, or that ends in a splice, shall be processed as if an additional new-line character were appended to the file.

**The ## operator**

A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a placemarker preprocessing token instead. For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token. **If the result begins with a sequence matching the syntax of universal-character-name, the behavior is undefined.** [Note: This determination
does not consider the replacement of universal-character-name s in translation phase 3. — end note]  If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of ##

**Future work**

The reader will have noticed that [cpp] has a few other undefined behaviors. This should equally be fixed, however this work is best left to someone with greater preprocessor expertise.