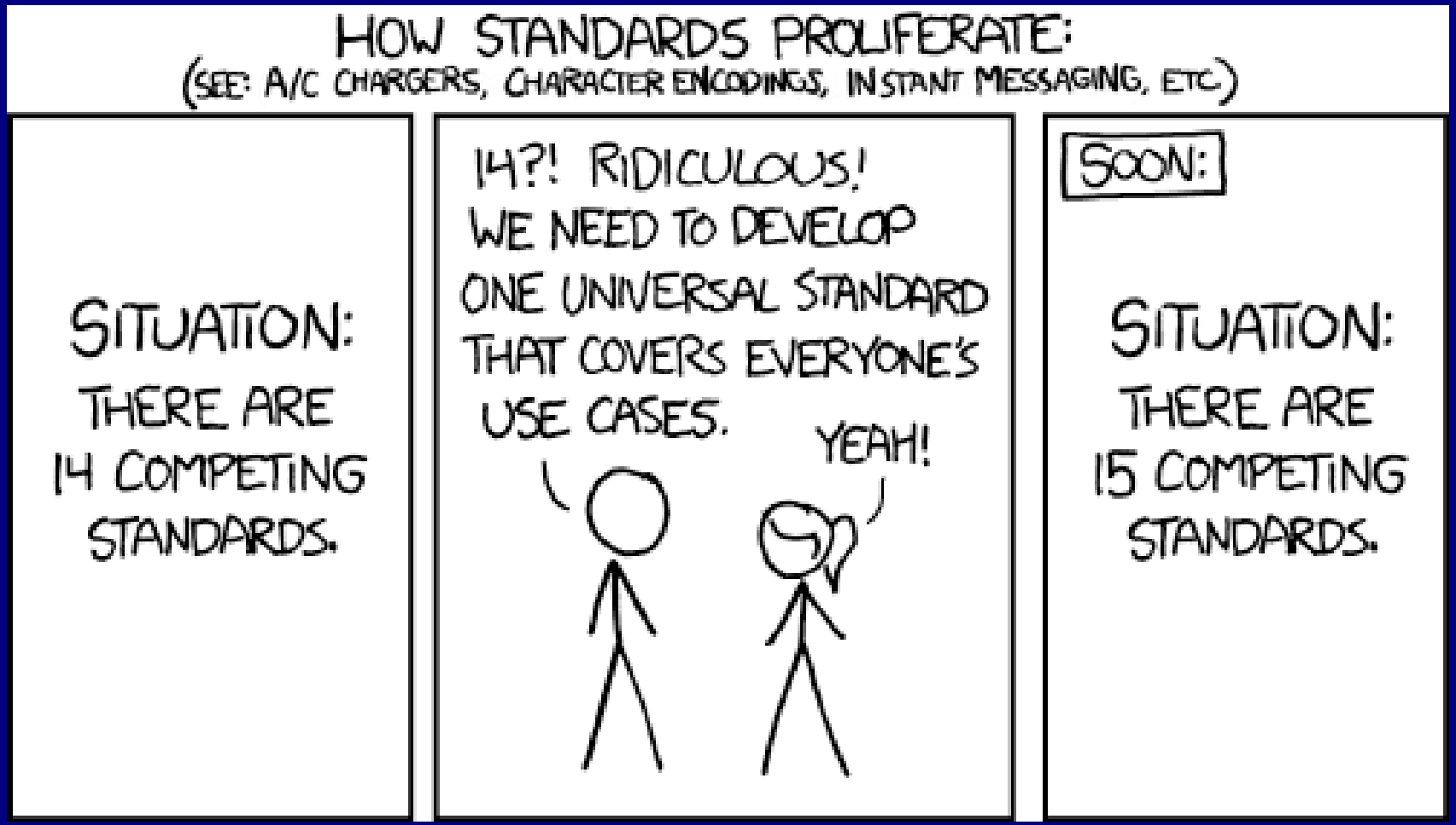
Overview of dxml and Lessons Learned

by Jonathan M Davis

All Text Formats Suck

All Text Formats Suck



std.xml

- std.xml has been considered substandard for years.
- No attempts to replace it have been completed.
  - 2010? attempt never resulted in public code.
  - GSoC student for std.experimental.xml went MIA.

Why Write a Solution in D?

Why Write a Solution in D?

- Dynamic arrays make D a fantastic fit for parsing.

```
struct DynamicArray(T)
{
    size_t length;
    T* ptr;
}
```

Design Principles / Constraints / Guidelines / Goals

- Range-based: It must accept arbitrary ranges of characters.
- It must take advantage of slicing semantics.
- If the input type is string, the output type must be string.
- No auto-decoding.
- Minimize heap allocations.

XML I Care About / My Bias

- Start Tags and End Tags
- Attributes on Start Tags
- Text Between Tags
- Comments

DOCTYPE Definition

- Provides information for validating the XML document.
- Allows for inserting XML from other XML documents.

```
<foo>&bar;</foo>
```

dxml does not support the DTD

- Slicing semantics are incompatible with DTD support.
- dxml cannot properly handle XML documents containing non-standard entity references.
- dxml either throws on such entity references or just validates that they're syntactically valid.

Initial Thoughts

- Build higher level parsers on lower level parsers.

Initial Thoughts

- Build higher level parsers on lower level parsers.
- What's the lowest level that makes sense?

Initial Thoughts

- Build higher level parsers on lower level parsers.
- What's the lowest level that makes sense?
- A range of entities.

Initial Thoughts

- Build higher level parsers on lower level parsers.
- What's the lowest level that makes sense?
- A range of entities.
- A range following the tree structure seemed desirable but ultimately not reasonably feasible.

Common XML Parser Types

- DOM: Document Object Model
- SAX: Simple API for XML
- StAX: Streaming API for XML

dxml is a Range-Based StAX / Pull Parser

"Streaming API for XML (StAX) is an application programming interface (API) to read and write XML documents, originating from the Java programming language community." - wikipedia

A StAX / Pull Parser essentially provides an iterator for linearly parsing an XML file.

Parsing Order

```
<root>
    <foo>
        <bar>some text</bar>
    </foo>
    <baz>other text</baz>
</root>
```

```
<root>
<foo>
<bar>
some text
</bar>
<baz>
other text
</baz>
</foo>
</root>
```

Tag Stack is Problematic

- Validating end tags requires a tag stack.
- Keeping track of the tag path requires a tag stack.
- Maintaining a tag stack requires allocating memory.
- Maintaining a tag stack becomes very problematic for the range API.

Why a Tag Stack Seemingly Makes Ranges Not Work

- A range of entities which know their path means maintaining a tag stack per entity.
- When a range is saved, it needs its own tag stack.
- Both of these meaning allocating memory, which really isn't acceptable.

dxml Became Cursor-Based

- save doesn't work, so it can't be a forward range.
- The entity can't be saved / copied, so an input range doesn't work.
- A cursor can provide access to the data without having to duplicate the tag stack.

Epiphanies

- Not being able to save *really* sucks.

Epiphanies

- Not being able to save *really* sucks.
- Usage showed that path was not useful.

Epiphanies

- Not being able to save *really* sucks.
- Usage showed that path was not useful.
- The XML only needs to be validated once.

Magic for Efficient Ranges

```
struct TagStack
{
    struct SharedState
    {
        Taken[] tags;
        Tuple!(Taken, TextPos)[] attrs;
        size_t maxEntities
    }

    SharedState* state;
    size_t entityCount;
    int depth;
}

alias Taken =
    typeof(takeExactly(byCodeUnit(R.init), 42));
```

Non-Exception Allocations in Parser

```
static create()
{
    TagStack tagStack;
    tagStack.state = new SharedState;
    tagStack.state.tags.reserve(10);
    tagStack.state.attrs.reserve(10);
    return tagStack;
}
```

dxml Modules

- dxml.dom
- dxml.parser
- dxml.util
- dxml.writer

dxml.parser

```d
auto range = parseXML(xml);
```

```d
auto range = parseXML!simpleXML(xml);
```

dxml.parser : Config

```
struct Config
{
    auto skipComments = SkipComments.no;

    auto skipPI = SkipPI.no;

    auto splitEmpty = SplitEmpty.no;

    auto throwOnEntityRef = ThrowOnEntityRef.yes;
}
```

dxml.parser : makeConfig

```
enum config1 = makeConfig(SplitEmpty.yes,
                          SkipComments.yes,
                          ThrowOnEntityRef.no);

enum config2 = makeConfig(SkipComments.yes,
                          ThrowOnEntityRef.no,
                          SplitEmpty.yes,

enum config3 = makeConfig(ThrowOnEntityRef.no,
                          SkipComments.yes);
```

Tracking Document Position

```
struct TextPos
{
    int line = 1;

    int col = 1;
}
```

Tracking Document Position

```
struct TextPos
{
    int line = 1;

    int col = 1;
}
```

```
enum PositionType
{
    lineAndCol,

    line,

    none,
}
```

Class Forward Ranges Are Terrible

- They are very inefficient to save.
- They don't have a valid init value.

```
@property auto save()
{
    // The init check nonsense is because of ranges whose init values blow
    // up when save is called (e.g. a range that's a class).
    auto retval = this;
    if(retval._name !is typeof(retval._name).init)
        retval._name = _name.save;
    if(retval._text.input !is typeof(retval._text.input).init)
        retval._text.input = _text.input.save;
    if(retval._savedText.input !is typeof(retval._savedText.input).init)
        retval._savedText.input = _savedText.input.save;
    return retval;
}
```

dxml.parser

```d
auto range = parseXML(xml);
```

```d
auto range = parseXML!simpleXML(xml);
```

EntityRange.Entity

```
EntityType type();

TextPos pos();

SliceOfR name();

auto attributes();

SliceOfR text();
```

```
static if(isDynamicArray!R || hasSlicing!R)
    alias SliceOfR = R;
else
    alias SliceOfR = typeof(takeExactly(R.init, 42));
```

enum EntityType

```
cdata

comment

elementStart

elementEnd

elementEmpty

pi

text
```

Sample XML

```
<?xml foobar sally?>
<?do something?>
<root>
    <!-- no comment -->
    <foo>some text</foo>
    <bar/>
    <baz></baz>
    <![CDATA[see data run]]>
</root>
```

dxml.parser - Config.init

```d
auto range = parseXML(xml);

// Skipped: <?xml foobar sally?>

// <?do something?>
assert(range.front.type == EntityType.pi);
assert(range.front.name == "do");
assert(range.front.text == "something");
range.popFront();
```

dxml.parser - Config.init

```d
// <root>
assert(range.front.type == EntityType.elementStart);
assert(range.front.name == "root");
range.popFront();

// <!-- no comment -->
assert(range.front.type == EntityType.comment);
assert(range.front.text == " no comment ");
range.popFront();
```

dxml.parser - Config.init

```d
// <foo>some text</foo>
assert(range.front.type == EntityType.elementStart);
assert(range.front.name == "foo");
range.popFront();

assert(range.front.type == EntityType.text);
assert(range.front.text == "some text");
range.popFront();

assert(range.front.type == EntityType.elementEnd);
assert(range.front.name == "foo");
range.popFront();
```

dxml.parser - Config.init

```
// <bar/>
assert(range.front.type == EntityType.elementEmpty);
assert(range.front.name == "bar");
range.popFront();

// <baz></baz>
assert(range.front.type == EntityType.elementStart);
assert(range.front.name == "baz");
range.popFront();

assert(range.front.type == EntityType.elementEnd);
assert(range.front.name == "baz");
range.popFront();
```

dxml.parser - Config.init

```
// <![CDATA[see data run]]>
assert(range.front.type == EntityType.cdata);
assert(range.front.text == "see data run");
range.popFront();

// </root>
assert(range.front.type == EntityType.elementEnd);
assert(range.front.name == "root");
range.popFront();
assert(range.empty);
```

dxml.parser : simpleXML

```
enum simpleXML = makeConfig(SkipComments.yes,
                            SkipPI.yes,
                            SplitEmpty.yes);
```

dxml.parser - simpleXML

```d
auto range = parseXML!simpleXML(xml);

// Skipped: <?xml foobar sally?>

// Skipped: <?do something?>
```

dxml.parser - simpleXML

```d
// <root>
assert(range.front.type == EntityType.elementStart);
assert(range.front.name == "root");
range.popFront();

// Skipped: <!-- no comment -->
```

dxml.parser - simpleXML

```d
// <foo>some text</foo>
assert(range.front.type == EntityType.elementStart);
assert(range.front.name == "foo");
range.popFront();

assert(range.front.type == EntityType.text);
assert(range.front.text == "some text");
range.popFront();

assert(range.front.type == EntityType.elementEnd);
assert(range.front.name == "foo");
range.popFront();
```

dxml.parser - simpleXML

```d
// <bar/>
assert(range.front.type == EntityType.elementStart);
assert(range.front.name == "bar");
range.popFront();

assert(range.front.type == EntityType.elementEnd);
assert(range.front.name == "bar");
range.popFront();

// <baz></baz>
assert(range.front.type == EntityType.elementStart);
assert(range.front.name == "baz");
range.popFront();

assert(range.front.type == EntityType.elementEnd);
assert(range.front.name == "baz");
range.popFront();
```

dxml.parser - simpleXML

```
// <![CDATA[see data run]]>
assert(range.front.type == EntityType.cdata);
assert(range.front.text == "see data run");
range.popFront();

// </root>
assert(range.front.type == EntityType.elementEnd);
assert(range.front.name == "root");
range.popFront();
assert(range.empty);
```

dxml.parser - Attributes

```d
auto xml = `<foo width="14" height='27'/>`;
auto range = parseXML(xml);
auto attrs = range.front.attributes;

assert(attrs.front.name == "width");
assert(attrs.front.value == "14");
attrs.popFront();

assert(attrs.front.name == "height");
assert(attrs.front.value == "27");
attrs.popFront();
assert(attrs.empty);
```

dxml.parser Helper Functions

- skipContents
- skipToPath
- skipToEntityType
- skipToParentEndTag

dxml.parser : skipToPath, skipContents

```
auto xml = "<superuser>\n" ~
           "    <foo>\n" ~
           "        <bar>\n" ~
           "            <baz>text</baz>\n" ~
           "        </bar>\n" ~
           "    </foo>\n" ~
           "</superuser>";

auto range = parseXML!simpleXML(xml);
range = range.skipToPath("foo/bar");
assert(range.front.type == EntityType.elementStart);
assert(range.front.name == "bar");

range = range.skipContents();
assert(range.front.type == EntityType.elementEnd);
assert(range.front.name == "bar");
```

Implementation Notes

- std.algorithm is used minimally.
- Keeping track of the column and line number is not straightforward with using std.algorithm.
- In some cases (e.g. startsWith), using a custom algorithm reduces how often the same data is iterated.

dxml.dom : DOMEntity

```
EntityType type();

TextPos pos();

SliceOfR name();

SliceOfR[] path(); // Not on EntityRange.Entity

auto attributes();

SliceOfR text();

DOMEntity[] children(); // Not on EntityRange.Entity
```

dxml.dom : parseDom

```
auto dom = parseDOM(xml);
```

```
auto dom = parseDOM!simpleXML(xml);
```

```
auto dom = parseDOM(parseXML(xml));
```

Sample XML

```
<root>
    <nothing></nothing>
    <foo>some text</foo>
    <bar/>
    <baz>more text</baz>
    <frobozz/>
</root>
```

dxml.dom : parseDom

```d
auto dom = parseDOM(xml);

// <root>
auto root = dom.children[0];
assert(root.type == EntityType.elementStart);
assert(root.name == "root");

// <baz>more text</baz>
auto baz = root.children[3];
assert(baz.type == EntityType.elementStart);
assert(baz.name == "baz");

auto text = baz.children[0];
assert(text.type == EntityType.text);
assert(text.text == "more text");
```

dxml.util - parsing

- decodeXML / asDecodedXML
- parseCharRef
- parseStdEntityRef
- stripIndent / withoutIndent

Standard Entity References

```
enum StdEntityRef
{
    amp = "&amp;", // &

    gt = "&gt;", // >

    lt = "&lt;", // <

    apos = "&apos;", // '

    quot = "&quot;", // "
}
```

dxml.util : decodeXML, asDecodedXML

```
auto xml = "a &lt; b &amp;&amp; foo &gt; bar";

assert(xml.decodeXML() == "a < b && foo > bar");

assert(equal(xml.asDecodedXML(),
             "a < b && foo > bar");
```

dxml.util : stripIndent

```
auto xml = "<carrot>\n" ~
           "    <code>\n" ~
           "        int getAnswer()\n" ~
           "        {\n" ~
           "            return 42;\n" ~
           "        }\n" ~
           "    </code>\n" ~
           "</carrot>";

auto range = parseXML!simpleXML(xml).drop(2);
assert(range.type == EntityType.text);
```

dxml.util : stripIndent

```
assert(range.front.text ==
        "\n" ~
        "        int getAnswer()\n" ~
        "        {\n" ~
        "            return 42;\n" ~
        "        }\n" ~
        "    ");

assert(range.front.text.stripIndent() ==
        "int getAnswer()\n" ~
        "{\n" ~
        "    return 42;\n" ~
        "}");
```

dxml.util - writing

- encodeText
- encodeAttr
- encodeCharRef

dxml.util : encodeText, encodeAttr

```
auto text = `& < > ' "`;

assert(equal(text.encodeText(),
             `&amp; &lt; > ' "`));

assert(equal(text.encodeAttr(),
             `&amp; &lt; > ' &quot;`));

assert(equal(text.encodeAttr!'\''(),
             `&amp; &lt; > &apos; "`));
```

dxml.writer : XMLWriter

```
auto writer = xmlWriter(appender!string());
writer.writeStartTag("root", Newline.no);

writer.openStartTag("foo");
writer.writeAttr("a", "42");
writer.closeStartTag();

writer.writeText("here is some text");

writer.writeEndTag("foo");

writer.writeEndTag("root");
```

dxml.writer : XMLWriter

```
assert(writer.output.data ==
       "<root>\n" ~
       `    <foo a="42">` ~ "\n" ~
       "        here is some text\n" ~
       "    </foo>\n" ~
       "</root>");
```

struct vs class

- XMLWriter needs to be a reference type.
- It's wasteful to allocate it by default.

struct vs class

- XMLWriter needs to be a reference type.
- It's wasteful to allocate it by default.
- With copying, assignment, and default initialization disabled, a struct can be treated as a reference type without allocating.

struct vs class

- XMLWriter needs to be a reference type.
- It's wasteful to allocate it by default.
- With copying, assignment, and default initialization disabled, a struct can be treated as a reference type without allocating.
- A class must be allocated, but scope and -dip1000 arguably would make it reasonable to use scope to avoid that allocation.

Benchmarking

111 MiB file 10 times / 1.1 MiB file 1000 times

dmd

dxml.parser	15 seconds	
dxm.dom	35 seconds	2.33x stax
std.xml	3 minutes 40 seconds	14.66x stax / 6.28x dom

ldc

dxml.parser	10 seconds	
dxm.dom	25 seconds	2.5x stax
std.xml	2 minutes 30 seconds	15x stax / 6x dom

Future

- More helper functions.
- Add support for writing from a DOM.
- Add configuration options for skipping some validation?
- Improve error messages.
- Optimizations / implementation improvements.
- Phobos?

Questions?