CEng 242 Homework 4

Due: 2^{nd} May 2004

This homework is pretty similar to your previous homework in ML. Instead of an abstract data type, you will implement a `Document` class to store, modify and dump XML documents hierarchically. There are two basic differences from the previous homework. First, you will implement a better XML syntax with attributes. Second, you will use a library called expat to import a XML file into your representation.

In the XML syntax, a tag can contain 0 or more attribute-value pairs as:
<tag attr1="val1" attr2="val2" ...>
So that information like style, name, cross references, links etc. can be put in the tag. Like

```
<xmldoc type="draft" author="Onur Sehitoglu">
    <chapter title="Introduction" style="header1">
        <section title="Problem Definition">
            <p>
            What is the problem?
            </p>
            <p>
            possible approaches
            </p>
        </section>
    </chapter>
</xmldoc>
```

You do not have to learn this syntax since expat library will take care of the syntax checking and parsing. Only thing you should now is you need to store the attribute-value pairs along the tags.

Different than the ML version, there is no limitation on the root element. You will have an invisible and initially empty root element and it may contain any number of tags and text.

Since C/C++ is weak in aggregate values (value constructors) you dint have the flexibility of list of tuples notation in ML for describing paths. Instead, we will use string (`const char *`) typed values containing a dash separated tag specifiers where each tag specifier consists of colon separated *tagname*`:`*count* pairs. For example:
`"xmldoc:1-chapter:1-section:1-p:2"`
string addresses the path where `"possible approaches"` text is contained.

You can use any data structure in private section of your implementation. As the interface, your class declaration should contain the following member functions in public section:

`Document()` Creates and empty document without any tags.

`Document(const char *)`
Gets a string parameter representing a filename, tries to parse the file using expat library and creates the resulting document. If parsing gives an error, empty document is constructed and error is reported as:
`cerr << "File:  " <<` *filename* `<< " has syntax error(s)\n";`

`Document & addText(const char *, const char *,int)`
`addText(`*Path*`,` *Text*`,` *Position*`)` will add the *Text* under the *Path* as the element in the *Position*. Returns reference to current object.

`Document & addText(const char *, const char *)`
This overloaded version adds the text as the last element of the path. You can implement this as a default negative parameter value for the *Position* parameter above.

`Document & addTag(const char *, const char *,int, ...)`
`addTag(`*Path*`,` *TagName*`,` *Position*`,` *[Attr , Val]*`*,` `NULL)` will add the *TagName* tag under the *Path* as the element in the *Position*. Following parameters should be `const char *` parameters which are attribute-value pairs of the tag. A `NULL` valued parameter ends the list. In order to implement this, you will use `stdarg.h` header and macros `va_start, va_arg, va_end`. See corresponding man pages for usage. The minimal usage without any attribute will be:
`Obj.addTag("xmldoc:1-chapter:2","section",3,NULL);`
Where the first chapter in the example can be added as:
`Obj.addTag("xmldoc:1","chapter",1,"title","Introduction", "style","header1",NULL);`

`Document & addDocument(const char *, const Document &, int)`
`addDocument(`*Path*`,` *Doc*`,` *Position*`)` will add all first level elements of document *Doc* under the *Path* as starting with the *Position*. Returns reference to current object.

`Document & addDocument(const char *, const Document &)`
This overloaded version appends the document content to the end of the path content. You can implement this as a default negative parameter value for the *Position* parameter above.

`Document & operator-=(const char *)`
This operator corresponds to `clearPath` function in the ML version. It gets a path parameter and deletes the last tag in the path together with its content on current object. Returns the reference to current object.

`Document operator[](const char *)`
This operator corresponds to `getPath` function in the ML version. It gets a path parameter and returns the sub-document under that path. The resulting document should contain only a single tag (last tag in the given path) in the first level.

`ostream & ostream::operator <<(const Document &)`
This operator is like `dumpXML` function in the ML version. However it is implemented as a member of the `ostream` class instead and in order to access class members, you may need to define this function as a friend function. As a result:
`cout << doc`
syntax should work. Each tag level is indented by 4 spaces. Tags are output as they are described in the XML syntax given above together with attribute-value pairs. Values should be enclosed in double quotes. For normalization, you should sort your attributes in alphabetical order. So the output will be like:
`<xmldoc Abc="..." ZZZ="..." abc="..." ccc="..." zzzz="...">`
regardless of the insertion order.

`void dumpTxt()`
`dumpTxt()` will output the text elements of the document on standard output. All tags, attributes and values are stripped. Carriage return is put after each text element and proper indentation with 4 spaces per nesting level is put into resulting string.

No find/replace function is required in this homework.

In functions with path parameter, if path is not a valid path, ignore the operation and produce output on standard error stream with:
`cerr << "Path: '" << Path << "' is not valid\n";`

In addition to this elements, you should implement all required public functions to guarantee safe assignment, pass by value, and return as a value operations.

When submitting this homework, you should submit 2 files packaged in a `tar.gz` file:

`hw4.h` This will contain only your `Document` class definition and nothing else. This class definition will only contain prototypes of the member functions and `operator<<` function. No inline definitions allowed in this file. No definitions other than the required definitions should be made public. You can define prototypes for private member functions.

`hw4.cpp` This file will include `hw4.h` and provide the implementations of all of the functions defined. If you happen to implement any auxiliary function outside of the class, define them as `static` so that they will not be available to linker. Do not put `main()` function here.

Than you can define a `hw4main.cpp` file as:

```
#include"hw4.h"

int main()
{
    Document a,b("./mydoc.xml");

    a.addTag("","xmldoc",1,"type","draft","author","Onur Sehitoglu",NULL);
    a.addTag("xmldoc:1","chapter",1,"title","Introduction",
                                "style","header1",NULL);
    a.addTag("xmldoc:1-chapter:1","section",1,
                                "title","Problem Definition",NULL);
    a.addTag("xmldoc:1-chapter:1-section:1","p",1,NULL);
    a.addTag("xmldoc:1-chapter:1-section:1","p",2,NULL);
    a.addText("xmldoc:1-chapter:1-section:1-p:2","possible approaches");
    a.addText("xmldoc:1-chapter:1-section:1-p:1","What is the problem?");

    Document c();

    c=a;
    cout << c;
    cout << b;

    a -= "xmldoc:1-chapter:1-section:1-p:1";

    cout << a;

    c.dumpTxt();

    c = a["xmldoc:1-chapter:1-section:1"];

    cout << c;

    a.dumpTxt();

    return 0;
}
```

You can use the following `Makefile` to build your code:

```
EXPATLIB=-L/usr/lib -lexpat
EXPATINC=-I/usr/include

hw4.o: hw4.h

.cpp.o:
        g++ -c $(EXPATINC) $<


hw4main: hw4.o hw4main.o
        g++ hw4.o hw4main.o -o hw4main $(EXPATLIB)
```

`Makefile` syntax requires the first characters in lines with `g++` to be a "TAB" character.

Cheaters will get 0 from all of the 6 homeworks.

EXPAT Library

Expat library home page is at `http://www.libexpat.org/` . You will only use a minimum functionality of this library. What library provides is to call user defined entry points at: entrance to a tag, end of the tag, and the text elements enclosed. In order to parse each file, you should create a parser by calling `XML_ParserCreate(const XML_Char *encoding);`.

Then you can introduce a pointer to a local variable or structure that will be passed to all function calls by the parser via `XML_SetUserData(XML_Parser p, void *userData);` call. With help of this call you may make a local variable available across the parser calls.

The next step is to introduce entry points that will be called by the parser. You will implement this functions and say `expat` library to call this functions via a call to `XML_SetElementHandler(XML_Parser p, XML_StartElementHandler start, XML_EndElementHandler end);` and `XML_SetCharacterDataHandler(XML_Parser p, XML_CharacterDataHandler charhndl)`. The first will set the handler for start of a tag and the former sets the handler for the text data enclosed in tags.

These handler functions you should implement have the following type definitions:

```
typedef void
(*XML_StartElementHandler)(void *userData,
                           const XML_Char *name,
                           const XML_Char **atts);
typedef void
(*XML_EndElementHandler)(void *userData,
                         const XML_Char *name);

typedef void
(*XML_CharacterDataHandler)(void *userData,
                            const XML_Char *s,
                            int len);
```

In all handlers, the first parameter is the user data pointer that you may define by `XML_SetUserData`. In start element handler the second parameter is the name of the tag and the third element is an array of strings terminated by NULL pointer. This array contains attribute-value sequences. In end element handler the second parameter is simply the tag to end.

In character data handler, the second parameter is the text enclosed. However it is not zero terminated so you should use length parameter to define the significant part of it. Another problem with this function is it does not give full text in a single call. A large text body can be parsed as subsequent calls to this handler. You should detect these subsequent calls and merge this strings as necessary.

Another trouble with character data handler is that it is called for all intermediate characters including spaces, tabs and carriage returns. In order to make it more useful, you should eliminate the heading and trailing spaces from text elements.

Following is a sample C code for expat parsing. Heading spaces are eliminated but trailing spaces are left as they are. No merging of text elements is made:

```
#include <stdio.h>
#include "expat.h"

struct State {
   int depth, inText;
};

static void
startElement(void *userData, const char *name, const char **atts)
{
  int i;
  struct State *sttptr  = (struct State *)userData;
  for (i = 0; i < sttptr->depth; i++)
    putchar('\t');
  puts(name);
  sttptr->depth += 1;
  sttptr->inText = 0;
}

static void
endElement(void *userData, const char *name)
{
  struct State *sttptr  = (struct State *)userData;
  sttptr->depth -= 1;
  sttptr->inText = 0;
}

static void
charElement(void *userData, const char *str, int len)
{
  int i=0,j;
  struct State *sttptr  = (struct State *)userData;

  if (! sttptr->inText) {
   for (i=0;i<len && (str[i] == ' ' ||
str[i] == '\n' || str[i] == '\t');i++);

if (i<len)
sttptr->inText = 1;
  }
  if (sttptr->inText) {
  for (j = 0; j < sttptr->depth; j++)
```

```
   putchar('\t');
  putchar('[');
  for (; i < len; i++)
    putchar(str[i]);
  putchar(']'); putchar('\n');
  }
}

int
main(int argc, char *argv[])
{
  char buf[BUFSIZ];
  XML_Parser parser = XML_ParserCreate(NULL);
  int done;
  int depth = 0;
  struct State state={ 0, 0};
  XML_SetUserData(parser, (void *)&state);
  XML_SetElementHandler(parser, startElement, endElement);
  XML_SetCharacterDataHandler(parser, charElement);
  do {
    size_t len = fread(buf, 1, sizeof(buf), stdin);
    done = len < sizeof(buf);
    if (XML_Parse(parser, buf, len, done) == XML_STATUS_ERROR) {
      fprintf(stderr,
              "%s at line %d\n",
              XML_ErrorString(XML_GetErrorCode(parser)),
              XML_GetCurrentLineNumber(parser));
      return 1;
    }
  } while (!done);
  XML_ParserFree(parser);
  return 0;
}
```