Project Report : XML CONVERTER

Project 1 | HUST | 20201

Vũ Tấn Khang - 20183561

Section 1 Introduction

1. Introduction to Extensible Markup Language

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The design goals of XML emphasize simplicity, generality, and usability across the Internet. Several schema systems exist to aid in the definition of XML-based languages, while programmers have developed many application programming interfaces (APIs) to aid the processing of XML data.

The key constructs in an XML file that are often encountered in day-to-day use are:

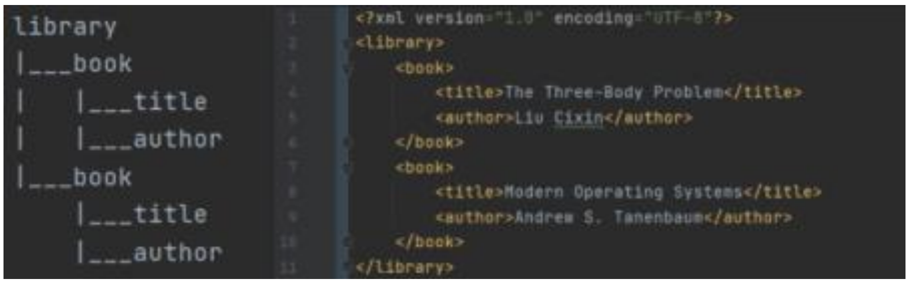
- **Tag**: A *tag* is a markup construct that begins with < and ends with >.
- **Element**: An *element* is a logical document component that either begins with a start-tag and ends with a matching end-tag or consists only of an empty-element tag. E.g. <greeting>Hello, world!</greeting>
- **Attribute**: An *attribute* is a markup construct consisting of a name-value pair that exists within a start-tag or empty-element tag. E.g. <step number="3">Connect A to B.</step>
- **XML declaration**: XML documents may begin with an *XML declaration* that describes some information about themselves. An example is <?xml version="1.0" encoding="UTF-8"?>.

2. XML Tree and Project Overview

Our goal is to extract objects from an XML file and writing it back to an output file (also an XML file). Therefore, we need a particular data structure to represent objects contain in the given file.

*XML documents have a hierachical structure and can conceptually be interpreted as a tree structure, called an **XML tree**.*

XML documents contain a root element (one that is the parent of all other elements). All elements in an XML document can contain sub elements, text and attributes. The tree represented by an XML document starts at the root element and branches to the lowest level of elements.



```
library
|___book
|    |___title
|    |___author
|___book
     |___title
     |___author
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<library>
    <book>
        <title>The Three-Body Problem</title>
        <author>Liu Cixin</author>
    </book>
    <book>
        <title>Modern Operating Systems</title>
        <author>Andrew S. Tanenbaum</author>
    </book>
</library>
```

1 - Fig.1. An XML document represents books contained in a library and, a tree that depict the structure of the file.

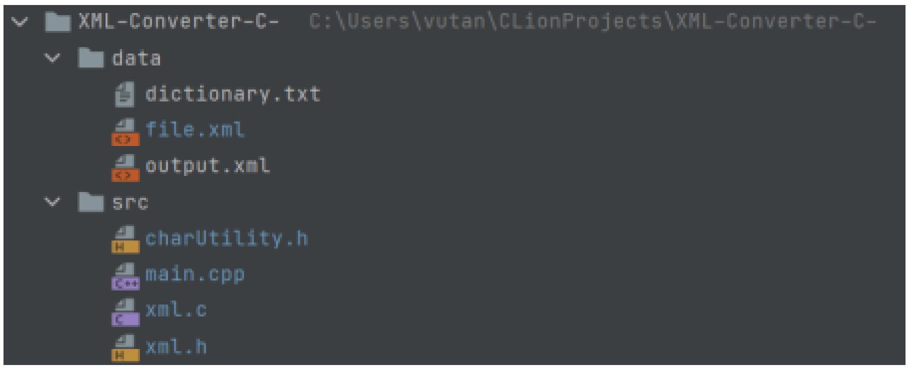
In this project, we use C as the primary programming language that handles the core implementation include both extracting objects from an XML file and converting all objects back into an XML file. The problem when using C in this project is that we need to allocate memory and free it before running the next iteration.

Beside C, C++ will be used to generate 10000 test cases continuously.

Section 2 Project Structure

Using the idea of XML tree, we will take the entire XML file, concatenate all the information into a string. After that, we will read and extract object' information and save to a root element.

There are two main parts of the project's source code. First, the core implementation that extracting (converting) objects from (to) an XML file (xml.h and xml.c). Second, main.cpp provide test cases and call functions from 'xml.h'. We also have an utility file named charUtility.h that handle some string manipulations.



```
XML-Converter-C-  C:\Users\vutan\CLionProjects\XML-Converter-C-
    data
        dictionary.txt
        file.xml
        output.xml
    src
        charUtility.h
        main.cpp
        xml.c
        xml.h
```

2 - Fig.2. Project Files

1. C/C++ Header File

The list below provides variables and functions used in two header files 'xml.h' and 'charUtility.h'.

'xml.h'

- typedef struct xmlElement
    - tag: char*
    - field: char*
    - next: struct xmlElement*

  - ◦ attrRoot: struct xmlElement* (the first attribute node in linked list)
  - ◦ attrNode: struct xmlElement* (the current attribute node in linked list)
  - ◦ elemRoot: struct xmlElement* (the first element node in linked list)
  - ◦ elemNode: struct xmlElement* (the current element node in linked list)
- typedef struct xmlLinkedList
  - ◦ previous: struct xmlLinkedList*
  - ◦ element: xmlElement*
- *function*: xmlElement* create();
  - ◦ "Initialize a new XML element (xmlElement)"
- *function*: __xmlElement(xmlElement* e, char* text, int l, int r);
  - ◦ "Get attribute and field if any of these exists"
- *function*: xmlElement* extract(char* xml);
  - ◦ "With a string (concatenate from the xml file) extract objects, save to root element"
- *function*: void convert(FILE* file, xmlElement* node, int depth);
  - ◦ "Take the root element, write to 'file' the output xml file"
- *function*: freeXmlElement(xmlElement* node);
  - ◦ "Free memory allocated for a given node"
- *function*: void exe();
  - ◦ "Read the XML file from ~/data/file.xml, write another XML file to ~/data/file.xml"

'charUtility.h': This file contains functions for handling char* manipulation

- *function*: char* substring(char* str, int l, int r);
- *function*: int stringLen(char* c);
- *function*: char* concat(char* x, char* y);
- *function*: char isSpace(char c);

2. C/C++ Source File

The program run 'main.cpp' and call **exe**() function in 'xml.c'. In main, using word from a dictionary file, we generate and XML file and save it into input file - file.xml in each iteration.

There are 10000 iterations in total, and each of them will be parsed by the function exe() above. First, the information from the input file will be concatenate into a string (*char) named 'text', then we extract objects to a root element by calling function **extract**(text). After we get the XML tree, we use

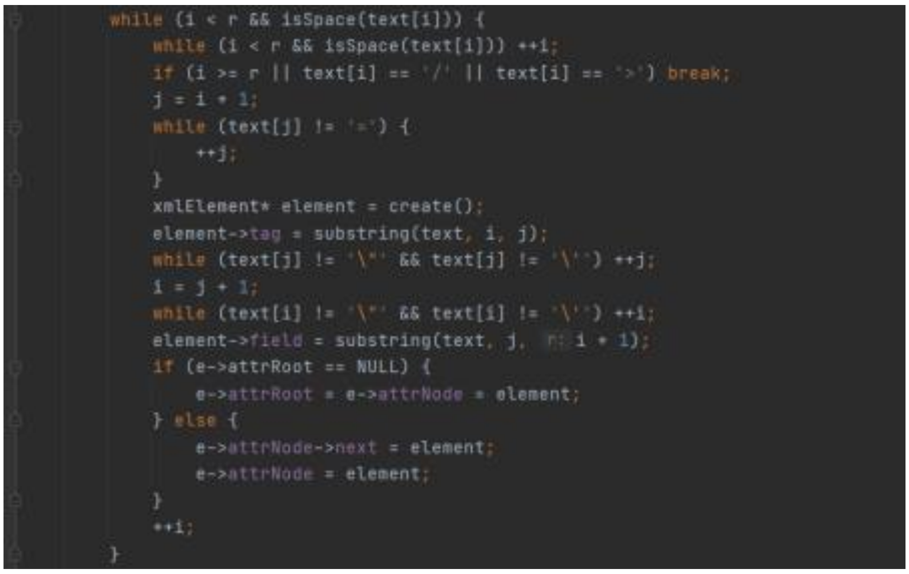
convert function to convert back all the information of objects to the output file (output.xml). And finally, free memory we allocated to the root element.

2.1 XML Element

In __XMLElement function, input includes a pointer type xmlElement, a string text (char*) and a pair of left, right position in the text. We want to get the attributes and field between a couple tag and save to the pointer xmlElement.

- First, extract the tag of input element (e), and then go to the end of this tag, see if there is any attribute.
- Second, read the field of the input element and save it to e->field.

Note: When getting information of a tag, we reuse another xmlElement. So, the tag of this element will become the name of the attribute, and the value of an attribute can be saved as the field of e.



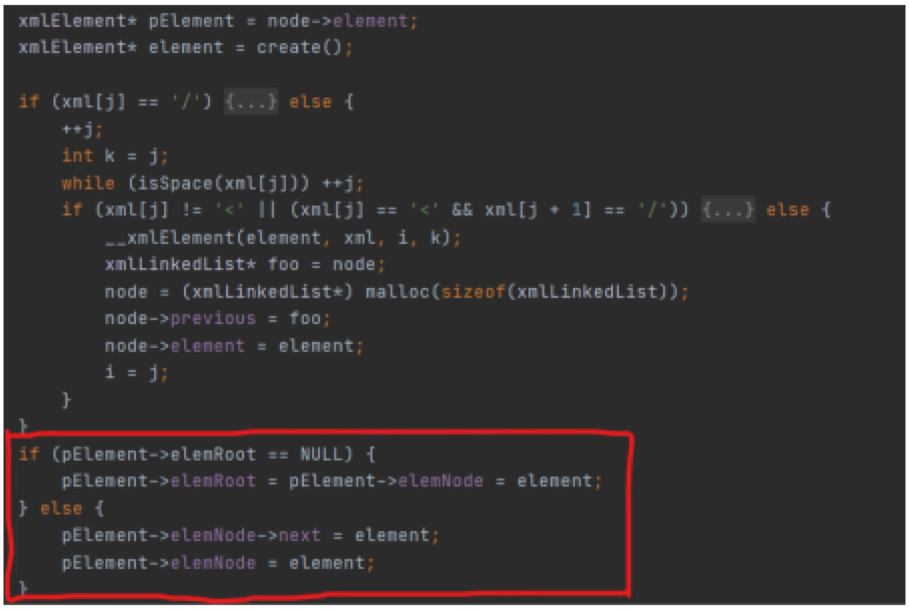
```
while (i < r && isSpace(text[i])) {
    while (i < r && isSpace(text[i])) ++i;
    if (i >= r || text[i] == '/' || text[i] == '>') break;
    j = i + 1;
    while (text[j] != '=') {
        ++j;
    }
    xmlElement* element = create();
    element->tag = substring(text, i, j);
    while (text[j] != '\"' && text[j] != '\'') ++j;
    i = j + 1;
    while (text[i] != '\"' && text[i] != '\'') ++i;
    element->field = substring(text, j, n: i + 1);
    if (e->attrRoot == NULL) {
        e->attrRoot = e->attrNode = element;
    } else {
        e->attrNode->next = element;
        e->attrNode = element;
    }
    ++i;
}
```

3 - Fig.3. Getting attributes by making use of an xmlElement

2.2 Extracting Objects

In detail, in extract function (xmlElement* extract(char* xml)), we will constructing an XML tree by going through these following steps:

- Create the root element - xmlElement*, create a **node** - xmlLinkedList* (node->element = root).
- Inside a while loop (**node** acts as a stack)
    - If there is a close-tag, delete the last element in node, continue
    - If there is an open-tag, push a new element (0) into node
    - At the end, connect (add edge between) parent and child element
- Free node and return root.



```
xmlElement* pElement = node->element;
xmlElement* element = create();

if (xml[j] == '/') {...} else {
    ++j;
    int k = j;
    while (isSpace(xml[j])) ++j;
    if (xml[j] != '<' || (xml[j] == '<' && xml[j + 1] == '/')) {...} else {
        __xmlElement(element, xml, i, k);
        xmlLinkedList* foo = node;
        node = (xmlLinkedList*) malloc(sizeof(xmlLinkedList));
        node->previous = foo;
        node->element = element;
        i = j;
    }
}
if (pElement->elemRoot == NULL) {
    pElement->elemRoot = pElement->elemNode = element;
} else {
    pElement->elemNode->next = element;
    pElement->elemNode = element;
}
```

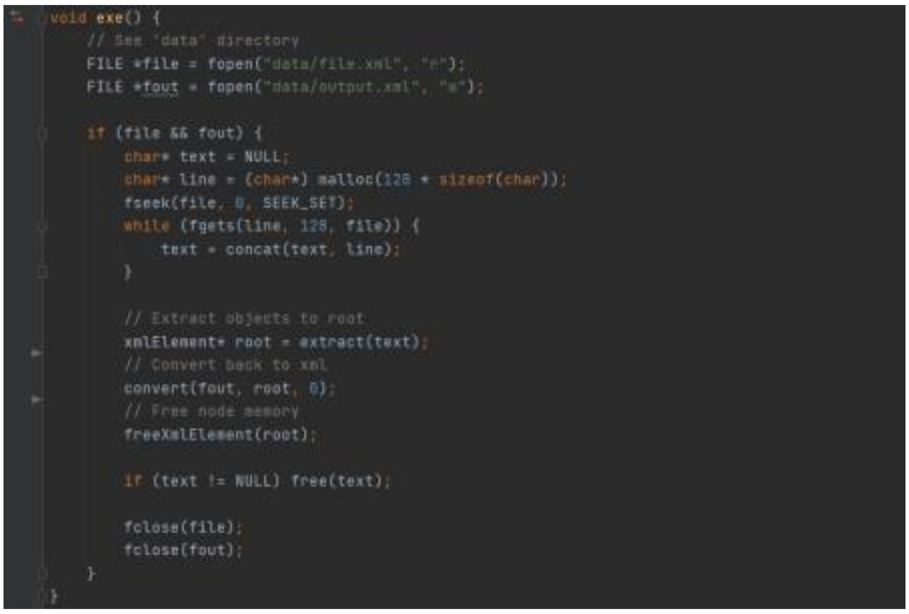
4 - Fig.3. Inside a while loop (extract function)

2.3 Convert Function and the 'main.cpp'

Finally, we use a recursive funtion 'convert' to print all the objects attach to the root element.

'main.cpp'

- *function*: void random_xml(int depth, string tab);
    - "Generate an XML file with the maximum depth of an XML element equals to 'depth'"
- extern "C" { #include "xml.h" }
- class: foo
    - "Call function exe() from xml.c"



```
void exe() {
    // See 'data' directory
    FILE *file = fopen("data/file.xml", "r");
    FILE *fout = fopen("data/output.xml", "w");

    if (file && fout) {
        char* text = NULL;
        char* line = (char*) malloc(128 * sizeof(char));
        fseek(file, 0, SEEK_SET);
        while (fgets(line, 128, file)) {
            text = concat(text, line);
        }

        // Extract objects to root
        xmlElement* root = extract(text);
        // Convert back to xml
        convert(fout, root, 0);
        // Free node memory
        freeXmlElement(root);

        if (text != NULL) free(text);

        fclose(file);
        fclose(fout);
    }
}
```

5 - Fig.4. Execution function (exe() from 'xml.c')

3. Data Folder

- dictionary.txt: contain words constructing input XML file (~/data/file.xml)
- file.xml: see function: void exe();
- output.xml: see function: void exe();

References

[1] Theory: XML, JetBrains Academy: https://hyperskill.org/learn/step/6901

[2] XML, Wikipedia: https://en.wikipedia.org/wiki/XML

[3] XML tree, Wikipedia: https://en.wikipedia.org/wiki/XML_tree