

Project 07

● Graded

1 Day, 22 Hours Late

Student

Ishan Mukherjee

Total Points

98 / 100 pts

Autograder Score

80.0 / 80.0

Passed Tests

Test 1

Test 2

Test 3

Question 2

Manual review

18 / 20 pts

✓ + 2 pts "busstop.cpp" header comment has description

✓ + 2 pts "busstop.cpp" header comment has name

+ 1 pt "busstop.cpp" header comment school, course, etc.

Commenting of "busstop.cpp"

✓ + 5 pts every function has a header comment, and some commenting within as appropriate

+ 3 pts some commenting / header comments, but not complete

+ 1 pt minimal commenting

+ 0 pts no commenting

✓ + 2 pts "busstops.cpp" header comment has description

✓ + 2 pts "busstops.cpp" header comment has name

+ 1 pt "busstops.cpp" header comment school, course, etc.

Commenting of "busstops.cpp"

✓ + 5 pts every function has a header comment, and some commenting within as appropriate

+ 3 pts some commenting / header comments, but not complete

+ 1 pt minimal commenting

+ 0 pts no commenting

Autograder Results

Autograder Output

```
*****
** Running lizard to analyze coding style, looking to see if functions **
** exceed 100 lines of code, which is considered too long... **
**
**
** We are also using lizard to identify nested loops, which are not **
** allowed... **
**
** If you see no warnings below, all is well. **
*****
```

```
*****
This is submission #2
Submitted @ 22:31 on 2024-3-3 (Chicago time)
```

Submission history:
Submission #1: score=0, submitted @ 21:56 on 2024-3-3 (Chicago time)

Total # of valid submissions so far: 1
of valid submissions since midnight: 1
of minutes since last valid submission: 35

You have 1 submission this 24-hr period.

```
*****
** Number of Submissions This Time Period **
*****
```

This is submission #2 in current time period

You are allowed a total of 4 submissions per 24-hr time period.

```
*****
```

```
*****
** Test Number: 1 **
```

```
** Test Input:
nu.osm
@
Mudd
$
```

```
** Your output (first 600 lines) **
** NU open street map **
```

Enter map filename>

of nodes: 15070

of buildings: 103

of bus stops: 12

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

14403: bus 201, Chicago Ave & Church, Southbound, NW corner, location (42.048, -87.679)

14692: bus 201, Chicago Ave & Davis, Southbound, NW corner, location (42.0464, -87.6797)

15318: bus 201, Chicago & Clark, Southbound, NW corner, location (42.0497, -87.6782)

15924: bus 201, Central & Sheridan, Eastbound, SW corner, location (42.064, -87.6773)

17295: bus 201, Sheridan & Foster, Southbound, SW corner, location (42.0539, -87.6774)

17296: bus 201, Sheridan & Foster, Northbound, East side, location (42.0542, -87.6771)

17301: bus 201, Sheridan & Lincoln, Southbound, SW corner, location (42.0614, -87.6773)

17302: bus 201, Chicago & Sheridan, Southbound, SW corner, location (42.0509, -87.6776)

18354: bus 201, Sheridan & Emerson, Northbound, East side, location (42.0523, -87.6771)

18355: bus 201, Sheridan & Haven, Northbound, East side, location (42.0576, -87.677)

18356: bus 201, Sheridan & Lincoln, Northbound, NE corner, location (42.0618, -87.677)

18357: bus 201, Sheridan & Haven, Southbound, NW corner, location (42.0581, -87.6771)

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

Seeley G. Mudd Science and Engineering Library

Address: 2233 Tech Drive

Building ID: 42703541

perimeter nodes: 14

Location: (42.0581, -87.6741)

Closest southbound bus stop:

18357: Sheridan & Haven, bus #201, NW corner, 0.154706 miles

vehicle #1808 on route 201 travelling Eastbound to arrive in 5 mins

Closest northbound bus stop:

18355: Sheridan & Haven, bus #201, East side, 0.155197 miles

vehicle #1177 on route 201 travelling Westbound to arrive in 22 mins

vehicle #1932 on route 201 travelling Westbound to arrive in 24 mins

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

** Done **

** Your program generated the correct outputs, **

** well done! The last step is to run valgrind, **

** which runs your program again to look for **

** memory/logic errors (we are ignoring memory **

** leaks due to the CURL library)... **

** Well done, no memory/logic errors! **

** End of Test 1 **

** Test Number: 2 **

** Test Input:

nu.osm

Segal

University Library

Tech

fred

@

\$

** Your output (first 600 lines) **

** NU open street map **

Enter map filename>

of nodes: 15070

of buildings: 103

of bus stops: 14

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

Segal Visitors Center/Parking Garage

Address: 1841 Sheridan Road

Building ID: 275851101

perimeter nodes: 34

Location: (42.0505, -87.6735)

Closest southbound bus stop:

17302: Chicago & Sheridan, bus #201, SW corner, 0.211051 miles

vehicle #1893 on route 201 travelling Eastbound to arrive in 10 mins

Closest northbound bus stop:

18354: Sheridan & Emerson, bus #201, East side, 0.223246 miles

<<no predictions available>>

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

Northwestern University Library

Address: 1970 Campus Drive

Building ID: 175187764

perimeter nodes: 268

Location: (42.0531, -87.6742)

Closest southbound bus stop:

17295: Sheridan & Foster, bus #201, SW corner, 0.170603 miles

vehicle #1890 on route 201 travelling Eastbound to arrive in 8 mins
vehicle #1922 on route 201 travelling Eastbound to arrive in 9 mins
vehicle #17 on route 201 travelling Eastbound to arrive in 11 mins

Closest northbound bus stop:

18354: Sheridan & Emerson, bus #201, East side, 0.159381 miles

<<no predictions available>>

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

Northwestern University Technological Institute

Address: 2145 Sheridan Road

Building ID: 35598594

perimeter nodes: 42

Location: (42.0578, -87.6759)

Closest southbound bus stop:

18357: Sherida & Have, bus #201, NW corner, 0.0642461 miles

vehicle #1808 on route 201 travelling Eastbound to arrive in 5 mins

Closest northbound bus stop:

18355: Sheridann & Havenn, bus #201, East side, 0.0595626 miles

vehicle #1177 on route 201 travelling Westbound to arrive in 22 mins

vehicle #1932 on route 201 travelling Westbound to arrive in 24 mins

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

No such building

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

12345: bus 201, Sheridan & Nothing, Southbound, Some corner, location (43.9509, -86.6776)

14403: bus 201, Chicago Ave & Church, Southbound, NW corner, location (42.048, -87.679)

14692: bus 201, Chicago Ave & Davis, Southbound, NW corner, location (42.0464, -87.6797)

15318: bus 201, Chicago & Clark, Southbound, NW corner, location (42.0497, -87.6782)

15924: bus 201, Central & Sheridan, Eastbound, SW corner, location (42.064, -87.6773)

17295: bus 201, Sheridan & Foster, Southbound, SW corner, location (42.0539, -87.6774)

17296: bus 201, Sheridn & Fosterr, Northbound, East side, location (42.0542, -87.6771)

17301: bus 201, Sheridan & Lincoln, Southbound, SW corner, location (42.0614, -87.6773)

17302: bus 201, Chicago & Sheridan, Southbound, SW corner, location (42.0509, -87.6776)

18354: bus 201, Sheridan & Emerson, Northbound, East side, location (42.0523, -87.6771)

18355: bus 201, Sheridann & Havenn, Northbound, East side, location (42.0576, -87.677)

18356: bus 201, Sheridan & Lincoln, Northbound, NE corner, location (42.0618, -87.677)

18357: bus 201, Sherida & Have, Southbound, NW corner, location (42.0581, -87.6771)

56789: bus 201, Chicago & Nothing, Northbound, Another corner, location (41.9509, -88.6776)

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

** Done **

** Your program generated the correct outputs, **

** well done! The last step is to run valgrind, **

** which runs your program again to look for **

** memory/logic errors (we are ignoring memory **
** leaks due to the CURL library)... **

** Well done, no memory/logic errors! **

** End of Test 2 **

** Test Number: 3 **

** Test Input:
nu.osm
@
FRED
Annie May
Mudd
Segal
Swift
The frank zizza center for mathematics
Tech
University Library
\$

** Your output (first 600 lines) **
** NU open street map **

Enter map filename>
of nodes: 15070
of buildings: 103
of bus stops: 12

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>
14403: bus 201, Chicago Ave & Church, Northbound, NW corner, location (42.048, -87.679)
14692: bus 201, Chicago Ave & Davis, Northbound, NW corner, location (42.0464, -87.6797)
15318: bus 201, Chicago & Clark, Northbound, NW corner, location (42.0497, -87.6782)
15924: bus 201, Central & Sheridan, Westbound, SW corner, location (42.064, -87.6773)
17295: bus 201, Sheridan & Foster, Northbound, SW corner, location (42.0539, -87.6774)
17296: bus 201, Sheridan & Foster, Southbound, East side, location (42.0542, -87.6771)
17301: bus 201, Sheridan & Lincoln, Northbound, SW corner, location (42.0614, -87.6773)
17302: bus 201, Chicago & Sheridan, Northbound, SW corner, location (42.0509, -87.6776)
18354: bus 201, Sheridan & Emerson, Southbound, East side, location (42.0523, -87.6771)
18355: bus 201, Sheridan & Haven, Southbound, East side, location (42.0576, -87.677)

18356: bus 201, Sheridan & Lincoln, Southbound, NE corner, location (42.0618, -87.677)
18357: bus 201, Sheridan & Haven, Northbound, NW corner, location (42.0581, -87.6771)

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>
No such building

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

Annie May Swift Hall

Address: 1920 Campus Drive

Building ID: 33908908

perimeter nodes: 11

Location: (42.0524, -87.6751)

Closest southbound bus stop:

18354: Sheridan & Emerson, bus #201, East side, 0.105091 miles
vehicle #1818 on route 201 travelling Westbound to arrive in 24 mins

Closest northbound bus stop:

17295: Sheridan & Foster, bus #201, SW corner, 0.156074 miles
vehicle #1893 on route 201 travelling Eastbound to arrive in 11 mins

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

Seeley G. Mudd Science and Engineering Library

Address: 2233 Tech Drive

Building ID: 42703541

perimeter nodes: 14

Location: (42.0581, -87.6741)

Closest southbound bus stop:

18355: Sheridan & Haven, bus #201, East side, 0.155197 miles
vehicle #1023 on route 201 travelling Westbound to arrive in 2 mins
vehicle #1933 on route 201 travelling Westbound to arrive in 7 mins

Closest northbound bus stop:

18357: Sheridan & Haven, bus #201, NW corner, 0.154706 miles
<<no predictions available>>

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

Segal Visitors Center/Parking Garage

Address: 1841 Sheridan Road

Building ID: 275851101

perimeter nodes: 34

Location: (42.0505, -87.6735)

Closest southbound bus stop:

18354: Sheridan & Emerson, bus #201, East side, 0.223246 miles
vehicle #1818 on route 201 travelling Westbound to arrive in 24 mins

Closest northbound bus stop:

17302: Chicago & Sheridan, bus #201, SW corner, 0.211051 miles
vehicle #1893 on route 201 travelling Eastbound to arrive in 10 mins

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

Annie May Swift Hall

Address: 1920 Campus Drive

Building ID: 33908908

perimeter nodes: 11

Location: (42.0524, -87.6751)

Closest southbound bus stop:

18354: Sheridan & Emerson, bus #201, East side, 0.105091 miles
vehicle #1818 on route 201 travelling Westbound to arrive in 24 mins

Closest northbound bus stop:

17295: Sheridan & Foster, bus #201, SW corner, 0.156074 miles
vehicle #1893 on route 201 travelling Eastbound to arrive in 11 mins

Swift Hall

Address: 2029 Sheridan Road

Building ID: 214618520

perimeter nodes: 25

Location: (42.0551, -87.6749)

Closest southbound bus stop:

17296: Sheridan & Foster, bus #201, East side, 0.128767 miles
vehicle #1818 on route 201 travelling Westbound to arrive in 18 mins

Closest northbound bus stop:

17295: Sheridan & Foster, bus #201, SW corner, 0.15194 miles
vehicle #1893 on route 201 travelling Eastbound to arrive in 11 mins

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

No such building

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

Northwestern University Technological Institute

Address: 2145 Sheridan Road

Building ID: 35598594

perimeter nodes: 42

Location: (42.0578, -87.6759)

Closest southbound bus stop:

18355: Sheridan & Haven, bus #201, East side, 0.0595626 miles
vehicle #1023 on route 201 travelling Westbound to arrive in 2 mins
vehicle #1933 on route 201 travelling Westbound to arrive in 7 mins

Closest northbound bus stop:

18357: Sheridan & Haven, bus #201, NW corner, 0.0642461 miles
<<no predictions available>>

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

Northwestern University Library

Address: 1970 Campus Drive

Building ID: 175187764

perimeter nodes: 268

Location: (42.0531, -87.6742)

Closest southbound bus stop:

18354: Sheridan & Emerson, bus #201, East side, 0.159381 miles
vehicle #1818 on route 201 travelling Westbound to arrive in 24 mins

Closest northbound bus stop:

17295: Sheridan & Foster, bus #201, SW corner, 0.170603 miles

vehicle #1893 on route 201 travelling Eastbound to arrive in 11 mins

Enter building name (partial or complete), or * to list, or @ for bus stops, or \$ to end>

** Done **

** Your program generated the correct outputs, **

** well done! The last step is to run valgrind, **

** which runs your program again to look for **

** memory/logic errors (we are ignoring memory **

** leaks due to the CURL library)... **

** Well done, no memory/logic errors! **

** End of Test 3 **

Excellent!

Test 1

Test 1: nu.osm @ and Mudd -- yay, output correct!

Test 2

Test 2: nu.osm then variety of inputs -- yay, output correct!

Test 3

Test 3: nu.osm then variety of inputs -- yay, output correct!

Submitted Files

```
1  /*building.cpp*/
2
3  //
4  // A building in the Open Street Map.
5  //
6  // Prof. Joe Hummel
7  // Northwestern University
8  // CS 211
9  //
10
11 #include <iostream>
12
13 #include "building.h"
14
15 using namespace std;
16
17
18 //
19 // constructor
20 //
21 Building::Building(long long id, string name, string streetAddr)
22 : ID(id), Name(name), StreetAddress(streetAddr)
23 {
24     //
25     // the proper technique is to use member initialization list above,
26     // in the same order as the data members are declared:
27     //
28     //this->ID = id;
29     //this->Name = name;
30     //this->StreetAddress = streetAddr;
31
32     // vector is default initialized by its constructor
33 }
34
35 //
36 // print
37 //
38 // prints information about a building --- id, name, etc. -- to
39 // the console. The function is passed the Nodes for searching
40 // purposes.
41 //
42 void Building::print(CURL* curl, Nodes& nodes, BusStops& busstops)
43 {
44     cout << this->Name << endl;
45     cout << "Address: " << this->StreetAddress << endl;
46     cout << "Building ID: " << this->ID << endl;
```

```

47  cout << "# perimeter nodes: " << this->NodeIDs.size() << endl;
48  pair<double, double> location = this->getLocation(nodes);
49  cout << "Location: (" << location.first << ", " << location.second << ")" << endl;
50  this->printAllClosestBusStops(curl, nodes, busstops);
51
52  // cout << "Nodes:" << endl;
53  // for (long long nodeid : this->NodeIDs)
54  // {
55  //   cout << " " << nodeid << ": ";
56
57  //   double lat = 0.0;
58  //   double lon = 0.0;
59  //   bool entrance = false;
60
61  //   bool found = nodes.find(nodeid, lat, lon, entrance);
62
63  //   if (found) {
64  //     cout << "(" << lat << ", " << lon << ")";
65
66  //     if (entrance)
67  //       cout << ", is entrance";
68
69  //     cout << endl;
70  //   }
71  //   else {
72  //     cout << "***NOT FOUND***" << endl;
73  //   }
74  // }//for
75 }
76
77 //
78 // printAllClosestBusStops
79 //
80 void Building::printAllClosestBusStops(CURL* curl, Nodes& nodes, BusStops& busstops)
81 {
82   this->printClosestBusStop(curl, nodes, busstops, "Southbound");
83   this->printClosestBusStop(curl, nodes, busstops, "Northbound");
84 }
85
86 //
87 // printClosestBusStop
88 //
89 void Building::printClosestBusStop(CURL* curl, Nodes& nodes, BusStops& busstops, string direction)
90 {
91   double this_lon = this->getLocation(nodes).first;
92   double this_lat = this->getLocation(nodes).second;
93
94   // lowercasing direction
95   string direction_lower = direction;

```

```

96 transform(direction_lower.begin(), direction_lower.end(), direction_lower.begin(),
97     [](unsigned char c){ return std::tolower(c); });
98
99 // printing closest bus stop message
100 cout << "Closest " << direction_lower << " bus stop:" << endl;
101
102 // initializing closest bus stop distance and index (in array of bus stops)
103 double closestDist = numeric_limits<double>::infinity();
104 int closestStopIndex = -1;
105
106 // iterating over all bus stops
107 for (int i = 0, n = busstops.MapBusStops.size(); i < n; i++)
108 {
109     double dist = distBetween2Points(this_lon, this_lat, busstops.MapBusStops[i].Location.first,
110     busstops.MapBusStops[i].Location.second);
111     // if right direction and closer than nearest bus stop found yet
112     if (busstops.MapBusStops[i].Direction == direction && dist < closestDist)
113     {
114         closestStopIndex = i;
115         closestDist = dist;
116     }
117 }
118 // print closest bus stop with distance
119 busstops.MapBusStops[closestStopIndex].printWithDist(closestDist);
120
121 // print bus predictions for closest bus stop
122 busstops.MapBusStops[closestStopIndex].printBusPredictions(curl);
123 }
124
125 //
126 // adds the given nodeid to the end of the vector.
127 //
128 void Building::add(long long nodeid)
129 {
130     this->NodeIDs.push_back(nodeid);
131 }
132
133 //
134 // gets the center (lat, lon) of the building based
135 // on the nodes that form the perimeter
136 //
137 pair<double, double> Building::getLocation(const Nodes& nodes)
138 {
139     // initializing variables used inside loop
140     double lat_sum, long_sum, cur_lat, cur_long;
141     lat_sum = long_sum = cur_lat = cur_long = 0.0;
142     bool entrance = false;
143

```

```
144 // loop through all of the building's nodes
145 for (long long id : this->NodeIDs)
146 {
147     if (!nodes.find(id, cur_lat, cur_long, entrance))
148         cout << "node not found" << endl;
149     lat_sum += cur_lat;
150     long_sum += cur_long;
151 }
152
153 // return avg latitude and longitude
154 size_t num_nodes = this->NodeIDs.size();
155 return make_pair(lat_sum / num_nodes, long_sum / num_nodes);
156 }
157
158
```

```
1  /*building.h*/
2
3  //
4  // A building in the Open Street Map.
5  //
6  // Prof. Joe Hummel
7  // Northwestern University
8  // CS 211
9  //
10
11 #pragma once
12
13 #include <string>
14 #include <vector>
15 #include <utility>
16 #include <cctype>
17 #include <limits>
18
19 #include "node.h"
20 #include "nodes.h"
21 #include "busstops.h"
22 #include "busstop.h"
23 #include "dist.h"
24 #include "curl_util.h"
25
26 using namespace std;
27
28
29 //
30 // Building
31 //
32 // Defines a campus building with a name (e.g. "Mudd"), a street
33 // address (e.g. "2233 Tech Dr"), and the IDs of the nodes that
34 // define the position / outline of the building.
35 //
36 // NOTE: the Name could be empty "", the HouseNumber could be
37 // empty, and the Street could be empty. Imperfect data.
38 //
39 class Building
40 {
41 public:
42     long long ID;
43     string Name;
44     string StreetAddress;
45     vector<long long> NodeIDs;
46 }
```



```
47 //
48 // constructor
49 //
50 Building(long long id, string name, string streetAddr);
51
52 //
53 // print
54 //
55 // prints information about a building --- id, name, etc. -- to
56 // the console. The function is passed the Nodes and BusStops for searching
57 // purposes.
58 //
59 void print(CURL* curl, Nodes& nodes, BusStops& busstops);
60
61 //
62 // printClosestBusStops
63 //
64 void printAllClosestBusStops(CURL* curl, Nodes& nodes, BusStops& busstops);
65
66 //
67 // printClosestBusStop
68 //
69 void printClosestBusStop(CURL* curl, Nodes& nodes, BusStops& busstops, string direction);
70
71 //
72 // adds the given nodeid to the end of the vector.
73 //
74 void add(long long nodeid);
75
76 //
77 // gets the center (lat, lon) of the building based
78 // on the nodes that form the perimeter
79 //
80 pair<double, double> getLocation(const Nodes& nodes);
81 };
82
```

```
1  /*buildings.cpp*/
2
3  //
4  // A collection of buildings in the Open Street Map.
5  //
6  // Prof. Joe Hummel
7  // Northwestern University
8  // CS 211
9  //
10
11 #include <iostream>
12 #include <string>
13 #include <vector>
14 #include <cassert>
15
16 #include "buildings.h"
17 #include "nodes.h"
18 #include "osm.h"
19 #include "tinyxml2.h"
20
21 using namespace std;
22 using namespace tinyxml2;
23
24
25 //
26 // readMapBuildings
27 //
28 // Given an XML document, reads through the document and
29 // stores all the buildings into the given vector.
30 //
31 void Buildings::readMapBuildings(XMLDocument& xmldoc)
32 {
33     XMLElement* osm = xmldoc.FirstChildElement("osm");
34     assert(osm != nullptr);
35
36     //
37     // Parse the XML document way by way, looking for university buildings:
38     //
39     XMLElement* way = osm->FirstChildElement("way");
40
41     while (way != nullptr)
42     {
43         const XMLAttribute* attr = way->FindAttribute("id");
44         assert(attr != nullptr);
45
46         //
```

```
47 // if this is a building, store info into vector:
48 //
49 if (osmContainsKeyValue(way, "building", "university"))
50 {
51     string name = osmGetKeyValue(way, "name");
52
53     string streetAddr = osmGetKeyValue(way, "addr:housenumber")
54         + " "
55         + osmGetKeyValue(way, "addr:street");
56
57     //
58     // create building object, then add the associated
59     // node ids to the object:
60     //
61     long long id = attr->Int64Value();
62
63     Building B(id, name, streetAddr);
64
65     XMLAttribute* nd = way->FirstChildElement("nd");
66
67     while (nd != nullptr)
68     {
69         const XMLAttribute* ndref = nd->FindAttribute("ref");
70         assert(ndref != nullptr);
71
72         long long id = ndref->Int64Value();
73
74         B.add(id);
75
76         // advance to next node ref:
77         nd = nd->NextSiblingElement("nd");
78     }
79
80     //
81     // add the building to the vector:
82     //
83     this->MapBuildings.push_back(B);
84 } //if
85
86 way = way->NextSiblingElement("way");
87 } //while
88
89 //
90 // done:
91 //
92 }
93
94 //
95 // print
```

```
96 //
97 // prints each building (id, name, address) to the console.
98 //
99 void Buildings::print()
100 {
101     for (const Building& B : this->MapBuildings) {
102         cout << B.ID << ": " << B.Name << ", " << B.StreetAddress << endl;
103     }
104 }
105
106 //
107 // findAndPrint
108 //
109 // Prints each building that contains the given name.
110 //
111 void Buildings::findAndPrint(CURL* curl, string name, Nodes& nodes, BusStops& busstops)
112 {
113     //
114     // find every building that contains this name:
115     //
116     bool found = false;
117
118     for (Building& B : this->MapBuildings)
119     {
120         if (B.Name.find(name) != string::npos) { // contains name:
121             B.print(curl, nodes, busstops);
122             found = true;
123         }
124     }
125
126     if (!found)
127         cout << "No such building" << endl;
128 }
129
130 //
131 // accessors / getters
132 //
133 int Buildings::getNumMapBuildings() {
134     return (int) this->MapBuildings.size();
135 }
136
```

```
1  /*buildings.h*/
2
3  //
4  // A collection of buildings in the Open Street Map.
5  //
6  // Prof. Joe Hummel
7  // Northwestern University
8  // CS 211
9  //
10
11 #pragma once
12
13 #include <iostream>
14 #include <string>
15 #include <vector>
16
17 #include "node.h"
18 #include "nodes.h"
19 #include "building.h"
20 #include "tinyxml2.h"
21 #include "curl_util.h"
22
23 using namespace std;
24 using namespace tinyxml2;
25
26
27 //
28 // Keeps track of all the buildings in the map.
29 //
30 class Buildings
31 {
32 public:
33     vector<Building> MapBuildings;
34
35     //
36     // readMapBuildings
37     //
38     // Given an XML document, reads through the document and
39     // stores all the buildings into the given vector.
40     //
41     void readMapBuildings(XMLDocument& xmldoc);
42
43     //
44     // print
45     //
46     // prints each building (id, name, address) to the console.
```

```
47 //
48 void print();
49
50 //
51 // findAndPrint
52 //
53 // Prints each building that contains the given name.
54 //
55 void findAndPrint(CURL* curl, string name, Nodes& nodes, BusStops& busstops);
56
57 //
58 // accessors / getters
59 //
60 int getNumMapBuildings();
61
62 };
63
64
65
```

```
1  /*busstop.cpp*/
2
3  //
4  // A bus stop in the Open Street Map.
5  //
6  // Ishan Mukherjee
7  //
8
9  #include "busstop.h"
10
11 using namespace std;
12
13 //
14 // constructor
15 //
16 BusStop::BusStop(long long id, double route, string name, string direction, string stoplocation,
17                  pair<double, double> location)
18     : ID(id), Route(route), Name(name), Direction(direction), StopLocation(stoplocation),
19       Location(location.first, location.second)
20 {}
21
22 //
23 // print
24 //
25 void BusStop::print()
26 {
27     cout << this->ID << ": ";
28     cout << "bus " << this->Route << ", ";
29     cout << this->Name << ", ";
30     cout << this->Direction << ", ";
31     cout << this->StopLocation << ", ";
32     cout << "location (" << this->Location.first << ", " << this->Location.second << ")" << endl;
33 }
34
35 //
36 // printWithDist
37 //
38 void BusStop::printWithDist(double dist)
39 {
40     cout << " " << this->ID << ": ";
41     cout << this->Name << ", ";
42     cout << "bus #" << this->Route << ", ";
43     cout << this->StopLocation << ", ";
44     cout << dist << " miles" << endl;
45 }
```

```

45 //
46 // printBusPredictions
47 //
48 void BusStop::printBusPredictions(CURL* curl)
49 {
50     string url = "http://ctabustracker.com/bustime/api/v2/getpredictions?
key=8yU8XjvHc5zZjtN4E8LyJsAn9&rt=" + to_string(this->Route) + "&stpid=" + to_string(this->ID) +
"&format=json";
51     string response = "";
52     if (!callWebServer(curl, url, response))
53     {
54         cout << " <<bus predictions unavailable, call failed>>" << endl;
55     }
56     else
57     {
58         // read in predictions from response
59         auto jsondata = json::parse(response);
60         auto bus_response = jsondata["bustime-response"];
61         auto predictions = bus_response["prd"];
62         // for each prediction (a map) in the list:
63         if (predictions.empty())
64             cout << " <<no predictions available>>" << endl;
65         for (auto& M : predictions) {
66             try
67             {
68                 // print prediction
69                 cout << " vehicle #" << stoi(M["vid"].get_ref<std::string&>()) << " ";
70                 cout << "on route " << stoi(M["rt"].get_ref<std::string&>()) << " ";
71                 cout << "travelling " << M["rtdir"].get_ref<std::string&>() << " ";
72                 cout << "to arrive in " << stoi(M["prcdtn"].get_ref<std::string&>()) << " mins" << endl;
73             }
74             catch (exception& e)
75             {
76                 cout << " error" << endl;
77                 cout << " malformed CTA response, prediction unavailable"
<< " (error: " << e.what() << ")" << endl;
78             }
79         }
80     }
81 }
82 }

```



```
1  /*busstop.h*/
2
3  //
4  // A bus stop in the Open Street Map.
5  //
6  // Ishan Mukherjee
7  //
8
9  #include <iostream>
10 #include <string>
11 #include <utility>
12 #include <stdexcept>
13 #include "curl_util.h"
14 #include "json.hpp"
15 using json = nlohmann::json;
16
17 #pragma once
18
19 using namespace std;
20
21 class BusStop
22 {
23 public:
24     long long ID;
25     long Route;
26     string Name;
27     string Direction;
28     string StopLocation;
29     pair<double, double> Location;
30
31     //
32     // constructor
33     //
34     BusStop(long long id, double route, string name, string direction, string stoplocation, pair<double,
double> location);
35
36     //
37     // print
38     //
39     void print();
40
41     //
42     // printWithDist
43     //
44     void printWithDist(double dist);
45
```

```
46 //  
47 // printWithDist  
48 //  
49 void printBusPredictions(CURL* curl);  
50 };
```

```
1  /*busstops.cpp*/
2
3  //
4  // A collection of buildings in the Open Street Map.
5  //
6  // Ishan Mukherjee
7  //
8
9  #include "busstops.h"
10
11 using namespace std;
12
13 //
14 // constructor
15 //
16 BusStops::BusStops(string filename)
17 {
18     ifstream file(filename);
19     if (!file)
20     {
21         cerr << "Error: unable to open bus stops csv " << filename << endl;
22     }
23
24     string line;
25     while (getline(file, line))
26     {
27         stringstream parser(line);
28         string id, route, name, direction, stop, loc_lat, loc_long;
29
30         // getting params for new bus stop
31         getline(parser, id, ',');
32         getline(parser, route, ',');
33         getline(parser, name, ',');
34         getline(parser, direction, ',');
35         getline(parser, stop, ',');
36         getline(parser, loc_lat, ',');
37         getline(parser, loc_long);
38
39         // creating new bus stop
40         this->MapBusStops.emplace_back(stoll(id, nullptr), stol(route, nullptr), name, direction, stop,
41 make_pair(stod(loc_lat, nullptr), stod(loc_long, nullptr)));
42     }
43     file.close();
44
45     sort(this->MapBusStops.begin(), this->MapBusStops.end(), [](BusStop a, BusStop b) {return a.ID <
```

```
    b.ID;});
46 }
47
48 //
49 // print
50 //
51 void BusStops::print()
52 {
53     for (BusStop& BS : this->MapBusStops)
54         BS.print();
55 }
56
57 //
58 // accessors / getters
59 //
60 int BusStops::getNumMapBusStops()
61 {
62     return (int) this->MapBusStops.size();
63 }
```

```
1  /*busstops.h*/
2
3  //
4  // A collection of bus stops in the Open Street Map.
5  //
6  // Ishan Mukherjee
7  //
8
9  #pragma once
10
11 #include <iostream>
12 #include <string>
13 #include <vector>
14 #include <fstream>
15 #include <sstream>
16 #include <algorithm>
17
18 #include "busstop.h"
19 #include "tinycl2.h"
20
21 using namespace std;
22 using namespace tinycl2;
23
24 class BusStops
25 {
26 public:
27     vector<BusStop> MapBusStops;
28
29     //
30     // constructor
31     //
32     BusStops(string filename);
33
34     //
35     // print
36     //
37     // prints each bus stop to the console.
38     //
39     void print();
40
41     //
42     // accessors / getters
43     //
44     int getNumMapBusStops();
45 };
```

```
1  /*curl_util.cpp*/
2
3  //
4  // CURL utility functions for calling a web server.
5  //
6  // Prof. Joe Hummel
7  // Northwestern University
8  // CS 211
9  //
10 // References:
11 //
12 // CURL library for internet access:
13 // https://everything.curl.dev/libcurl
14 //
15
16 #include <iostream>
17 #include <fstream>
18 #include <string>
19
20 #include "curl_util.h"
21
22 using namespace std;
23
24
25 //
26 // CURL_callback:
27 //
28 // This function is called when the CURL library receives a response
29 // from the web site. The function appends the contents of the response
30 // into the output, returning the total # of characters copied.
31 //
32 static size_t CURL_callback(void* contents, size_t size, size_t nmemb, std::string* output) {
33
34     size_t total_size = size * nmemb;
35
36     output->append((char*)contents, total_size);
37
38     return total_size;
39 }
40
41
42 //
43 // callWebServer:
44 //
45 // Given a URL, calls the web server attached to this URL and
46 // returns true if the web server responded, and false if not
```

```

47 // (false is also returned if there are problems with the CURL
48 // pointer that is passed). Note that the curl pointer should
49 // have been returned by a call to curl_easy_init().
50 //
51 // If true is returned, the response parameter will be set
52 // to the data returned by the server. If false is returned,
53 // response is unchanged.
54 //
55 // #define OFFLINE to test with offline saved data
56 // #define SAVE_ONLINE_RESPONSES if you want to save online data
57 // to use later when offline
58 //
59 static string getURLParam(string url, string key)
60 {
61     auto pos = url.find(key);
62     if (pos == string::npos)
63         return "-1";
64
65     auto endpos = url.find("&", pos);
66     if (endpos == string::npos)
67         return "-1";
68
69     size_t start = pos + key.length();
70     size_t len = endpos - start;
71     return url.substr(start, len);
72 }
73
74 bool callWebServer(CURL* curl, string url, string& response)
75 {
76     //
77     // URL format:
78     //
79     // http://ctabustracker.com/bustime/api/v2/getpredictions?
key=89dj2he89d8j3j3ksjhdue93j&rt=20&stpid=456&format=json
80     //
81
82     #ifdef OFFLINE
83     //
84     // We are offline, so return data saved from online calls:
85     //
86     string route = getURLParam(url, "rt=");
87     string stopid = getURLParam(url, "stpid=");
88
89     string filename = "cta-response-route-";
90     filename = filename + route + "-stopid-";
91     filename = filename + stopid + ".cta";
92
93     ifstream infile(filename);
94     if (!infile.is_open()) {

```

```
95     cout << "***ERROR: in callWebServer, we are running offline..." << endl;
96     cout << "***ERROR: you called with rt=" << route << ",stpid=" << stopid << endl;
97     cout << "***ERROR: no data is available for this route/stop" << endl;
98
99     return false;
100 }
101
102 string line;
103 response = "";
104
105 while (getline(infile, line))
106 {
107     response += line;
108     response += '\n';
109 }
110
111 infile.close();
112
113 return true;
114
115 #else
116 //
117 // call the online web service:
118 //
119 curl_easy_reset(curl);
120
121 int rc1 = curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
122 int rc2 = curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, CURL_callback);
123 int rc3 = curl_easy_setopt(curl, CURLOPT_WRITEDATA, &response);
124
125 if (rc1 != CURLE_OK || rc2 != CURLE_OK || rc3 != CURLE_OK)
126     return false;
127
128 //
129 // call web server, which triggers call to callback function when the
130 // response arrives:
131 //
132 int rc = curl_easy_perform(curl);
133
134 if (rc != CURLE_OK)
135     return false;
136
137 //
138 // if get here, it worked:
139 //
140 #ifndef SAVE_ONLINE_RESPONSES
141     string route = getURLParam(url, "rt=");
142     string stopid = getURLParam(url, "stpid=");
143
```



```
144     string filename = "cta-response-route-";
145     filename = filename + route + "-stopid-";
146     filename = filename + stopid + ".cta";
147
148     ofstream outfile(filename);
149     outfile << response << endl;
150     outfile.close();
151 #endif
152
153     return true;
154
155 #endif
156
157 }
158
159
```

```
1  /*curl_util.h*/
2
3  //
4  // CURL utility functions for calling a web server.
5  //
6  // Prof. Joe Hummel
7  // Northwestern University
8  // CS 211
9  //
10 // References:
11 //
12 // CURL library for internet access:
13 // https://everything.curl.dev/libcurl
14 //
15
16 #pragma once
17
18 #include <iostream>
19 #include <string>
20
21 #include <curl/curl.h>
22
23 using namespace std;
24
25 //
26 // callWebServer:
27 //
28 // Given a URL, calls the web server attached to this URL and
29 // returns true if the web server responded, and false if not
30 // (false is also returned if there are problems with the CURL
31 // pointer that is passed). Note that the curl pointer should
32 // have been returned by a call to curl_easy_init().
33 //
34 // If true is returned, the response parameter will be set
35 // to the data returned by the server. If false is returned,
36 // response is unchanged.
37 //
38 bool callWebServer(CURL* curl, string url, string& response);
39
```


```
1  /*dist.cpp*/
2
3  //
4  // Computes the distance between 2 positions, given in
5  // (latitude, longitude) coordinates.
6  //
7  // Prof. Joe Hummel
8  // Northwestern University
9  // CS 211
10 //
11
12 #include <iostream>
13 #include <cmath>
14
15 #include "dist.h"
16
17 using namespace std;
18
19
20 //
21 // DistBetween2Points
22 //
23 // Returns the distance in miles between 2 points (lat1, lon1) and
24 // (lat2, lon2). Latitudes are positive above the equator and
25 // negative below; longitudes are positive heading east of Greenwich
26 // and negative heading west. Example: Chicago is (41.88, -87.63).
27 //
28 // Reference: chatGPT using haversine formula
29 //
30 // static constexpr double M_PI = 3.141592653589;
31
32 static double toRadians(double degrees)
33 {
34     return (degrees * 3.141592653589 / 180.0);
35 }
36
37 double distBetween2Points(double lat1, double lon1, double lat2, double lon2)
38 {
39     const double R = 6371; // Earth's radius in kilometers
40
41     double dLat = toRadians(lat2 - lat1);
42     double dLon = toRadians(lon2 - lon1);
43
44     double a = sin(dLat / 2) * sin(dLat / 2) +
45             cos(toRadians(lat1)) * cos(toRadians(lat2)) *
46             sin(dLon / 2) * sin(dLon / 2);
```

```

47
48 double c = 2 * atan2(sqrt(a), sqrt(1 - a));
49
50 double dist_in_km = R * c;
51
52 double dist_in_miles = dist_in_km * 0.6213711922;
53
54 return dist_in_miles;
55 }
56

```

▼ dist.h

 Download

```

1  /*dist.h*/
2
3  //
4  // Computes the distance between 2 positions, given in
5  // (latitude, longitude) coordinates.
6  //
7  // Prof. Joe Hummel
8  // Northwestern University
9  // CS 211
10 //
11
12 #pragma once
13
14
15 //
16 // DistBetween2Points
17 //
18 // Returns the distance in miles between 2 points (lat1, long1) and
19 // (lat2, long2). Latitudes are positive above the equator and
20 // negative below; longitudes are positive heading east of Greenwich
21 // and negative heading west. Example: Chicago is (41.88, -87.63).
22 //
23 // Reference: chatGPT using haversine formula
24 //
25 double distBetween2Points(double lat1, double lon1, double lat2, double lon2);
26
27

```

▼ json.hpp

 Download

1 Large file hidden. You can download it using the button above.

```
1  /*main.cpp*/
2
3  //
4  // Program to input Nodes (positions) and Buildings from
5  // an Open Street Map file.
6  //
7  // Prof. Joe Hummel
8  // Northwestern University
9  // CS 211
10 //
11
12 #include <iostream>
13 #include <string>
14
15 #include "building.h"
16 #include "buildings.h"
17 #include "node.h"
18 #include "nodes.h"
19 #include "busstop.h"
20 #include "busstops.h"
21 #include "osm.h"
22 #include "tinycl2.h"
23 #include "curl_util.h"
24
25 using namespace std;
26 using namespace tinycl2;
27
28
29 //
30 // main
31 //
32 int main()
33 {
34     XMLDocument xmldoc;
35     Nodes nodes;
36     Buildings buildings;
37     BusStops busstops("bus-stops.txt");
38
39     cout << "*** NU open street map ***" << endl;
40
41     CURL* curl = curl_easy_init();
42     if (curl == nullptr) {
43         cout << "***ERROR:" << endl;
44         cout << "***ERROR: unable to initialize curl library" << endl;
45         cout << "***ERROR:" << endl;
46         return 0;
```

```
47 }
48
49 string filename;
50
51 cout << endl;
52 cout << "Enter map filename> " << endl;
53 getline(cin, filename);
54
55 //
56 // 1. load XML-based map file
57 //
58 if (!osmLoadMapFile(filename, xmldoc))
59 {
60     // failed, error message already output
61     return 0;
62 }
63
64 //
65 // 2. read the nodes, which are the various known positions on the map:
66 //
67 nodes.readMapNodes(xmldoc);
68
69 //
70 // 3. read the university buildings:
71 //
72 buildings.readMapBuildings(xmldoc);
73
74 //
75 // 4. stats
76 //
77 cout << "# of nodes: " << nodes.getNumMapNodes() << endl;
78 cout << "# of buildings: " << buildings.getNumMapBuildings() << endl;
79 cout << "# of bus stops: " << busstops.getNumMapBusStops() << endl;
80
81 //
82 // 5. now let the user for search for 1 or more buildings:
83 //
84 while (true)
85 {
86     string name;
87
88     cout << endl;
89     cout << "Enter building name (partial or complete), or * to list, or @ for bus stops, or $ to end> " <<
endl;
90
91     getline(cin, name);
92
93     if (name == "$") {
94         break;
```

```
95     }
96     else if (name == "**") {
97         buildings.print();
98     }
99     else if (name == "@")
100         busstops.print();
101     else {
102         buildings.findAndPrint(curl, name, nodes, busstops);
103     }
104
105 }//while
106
107 //
108 // done:
109 //
110 curl_easy_cleanup(curl);
111 cout << endl;
112 cout << "*** Done ***" << endl;
113
114 return 0;
115 }
116
```

```
1 build:
2     rm -f ./a.out
3     g++ -std=c++17 -g -Wall main.cpp building.cpp buildings.cpp node.cpp nodes.cpp busstop.cpp
busstops.cpp dist.cpp curl_util.cpp osm.cpp tinyxml2.cpp -Wno-unused-variable -Wno-unused-
function -lcurl
4
5 build-offline:
6     rm -f ./a.out
7     g++ -std=c++17 -g -Wall -DOFFLINE main.cpp building.cpp buildings.cpp node.cpp nodes.cpp
busstop.cpp busstops.cpp dist.cpp curl_util.cpp osm.cpp tinyxml2.cpp -Wno-unused-variable -Wno-
unused-function -lcurl
8
9 build-online-save:
10    rm -f ./a.out
11    g++ -std=c++17 -g -Wall -DSAVE_ONLINE_RESPONSES main.cpp building.cpp buildings.cpp
node.cpp nodes.cpp busstop.cpp busstops.cpp dist.cpp curl_util.cpp osm.cpp tinyxml2.cpp -Wno-
unused-variable -Wno-unused-function -lcurl
12
13 run:
14    ./a.out
15
16 valgrind:
17    rm -f ./a.out
18    g++ -std=c++17 -g -Wall main.cpp building.cpp buildings.cpp node.cpp nodes.cpp busstop.cpp
busstops.cpp dist.cpp curl_util.cpp osm.cpp tinyxml2.cpp -Wno-unused-variable -Wno-unused-
function -lcurl
19    valgrind --tool=memcheck --leak-check=full ./a.out
20
21 clean:
22    rm -f ./a.out
23
24 submit:
25    /home/cs211/w2024/tools/project07 submit *.cpp *.h *.hpp
26
```



```
1  /*node.cpp*/
2
3  //
4  // A node / position in the Open Street Map.
5  //
6  // Prof. Joe Hummel
7  // Northwestern University
8  // CS 211
9  //
10
11 #include "node.h"
12
13 using namespace std;
14
15
16 //
17 // constructor
18 //
19 Node::Node(long long id, double lat, double lon, bool entrance)
20 : ID(id), Lat(lat), Lon(lon), IsEntrance(entrance)
21 {
22     //
23     // the proper technique is to use member initialization list above,
24     // in the same order as the data members are declared:
25     //
26     //this->ID = id;
27     //this->Lat = lat;
28     //this->Lon = lon;
29     //this->IsEntrance = entrance;
30
31     Node::Created++;
32 }
33
34 //
35 // copy constructor:
36 //
37 Node::Node(const Node& other)
38 {
39     this->ID = other.ID;
40     this->Lat = other.Lat;
41     this->Lon = other.Lon;
42     this->IsEntrance = other.IsEntrance;
43
44     Node::Copied++;
45 }
46
```

```
47 //
48 // accessors / getters
49 //
50 long long Node::getID() const {
51
52     Node::CallsToGetID++;
53
54     return this->ID;
55 }
56
57 double Node::getLat() const {
58     return this->Lat;
59 }
60
61 double Node::getLon() const {
62     return this->Lon;
63 }
64
65 bool Node::getIsEntrance() const {
66     return this->IsEntrance;
67 }
68
69 int Node::getCallsToGetID() {
70     return Node::CallsToGetID;
71 }
72
73 int Node::getCreated() {
74     return Node::Created;
75 }
76
77 int Node::getCopied() {
78     return Node::Copied;
79 }
80
```

```
1  /*node.h*/
2
3  //
4  // A node / position in the Open Street Map.
5  //
6  // Prof. Joe Hummel
7  // Northwestern University
8  // CS 211
9  //
10
11 #pragma once
12
13 //
14 // Node:
15 //
16 // A node is a point on the map, with a unique ID and the position
17 // in GPS (lat, lon) terms. A node may also track other information,
18 // in particular whether this node denotes the entrance to a
19 // building.
20 //
21 class Node
22 {
23 private:
24     long long ID;
25     double Lat;
26     double Lon;
27     bool IsEntrance;
28
29     //
30     // These are static / class / singleton variables
31     // so that we can collect some statistics on how
32     // many times getID( ) is called, how many nodes
33     // are created, and how many are copied:
34     //
35     inline static int CallsToGetID = 0;
36     inline static int Created = 0;
37     inline static int Copied = 0;
38
39 public:
40     //
41     // constructor
42     //
43     Node(long long id, double lat, double lon, bool entrance);
44
45     //
46     // copy constructor:
```

```
47 //
48 Node(const Node& other);
49
50 //
51 // accessors / getters
52 //
53 long long getID() const;
54 double getLat() const;
55 double getLon() const;
56 bool getIsEntrance() const;
57
58 static int getCallsToGetID();
59 static int getCreated();
60 static int getCopied();
61
62 };
63
```

```
1  /*nodes.cpp*/
2
3  //
4  // A collection of nodes in the Open Street Map.
5  //
6  // Prof. Joe Hummel
7  // Northwestern University
8  // CS 211
9  //
10 // References:
11 //
12 // TinyXML:
13 // files: https://github.com/leethomason/tinyxml2
14 // docs: http://leethomason.github.io/tinyxml2/
15 //
16 // OpenStreetMap: https://www.openstreetmap.org
17 // OpenStreetMap docs:
18 // https://wiki.openstreetmap.org/wiki/Main\_Page
19 // https://wiki.openstreetmap.org/wiki/Map\_Features
20 // https://wiki.openstreetmap.org/wiki/Node
21 // https://wiki.openstreetmap.org/wiki/Way
22 // https://wiki.openstreetmap.org/wiki/Relation
23 //
24
25 #include <iostream>
26 #include <string>
27 #include <map>
28 #include <utility>
29 #include <algorithm>
30 #include <cassert>
31
32 #include "nodes.h"
33 #include "osm.h"
34 #include "tinyxml2.h"
35
36 using namespace std;
37 using namespace tinyxml2;
38
39
40 //
41 // readMapNodes
42 //
43 // Given an XML document, reads through the document and
44 // stores all the nodes into the given vector. Each node
45 // is a point on the map, with a unique id along with
46 // (lat, lon) position. Some nodes are entrances to buildings,
```

```

47 // which we capture as well.
48 //
49 void Nodes::readMapNodes(XMLDocument& xmldoc)
50 {
51     XMLElement* osm = xmldoc.FirstChildElement("osm");
52     assert(osm != nullptr);
53
54     //
55     // Parse the XML document node by node:
56     //
57     XMLElement* node = osm->FirstChildElement("node");
58
59     while (node != nullptr)
60     {
61         const XMLAttribute* attrId = node->FindAttribute("id");
62         const XMLAttribute* attrLat = node->FindAttribute("lat");
63         const XMLAttribute* attrLon = node->FindAttribute("lon");
64
65         assert(attrId != nullptr);
66         assert(attrLat != nullptr);
67         assert(attrLon != nullptr);
68
69         long long id = attrId->Int64Value();
70         double latitude = attrLat->DoubleValue();
71         double longitude = attrLon->DoubleValue();
72
73         //
74         // is this node an entrance? Check for a
75         // standard entrance, the main entrance, or
76         // one-way entrance.
77         //
78         bool entrance = false;
79
80         if (osmContainsKeyValue(node, "entrance", "yes") ||
81             osmContainsKeyValue(node, "entrance", "main") ||
82             osmContainsKeyValue(node, "entrance", "entrance"))
83         {
84             entrance = true;
85         }
86
87         //
88         // Add node to vector:
89         //
90         // The most concise way is using map's [] operator, but
91         // this requires a default constructor, which is bad design;
92         //
93         // Node N(id, latitude, longitude, entrance);
94         // this->MapNodes[id] = N;
95         //

```

```
96 // The insert() function is better, but creates an object
97 // then copies into the map:
98 //
99 // Node N(id, latitude, longitude, entrance);
100 // this->MapNodes.insert(make_pair(id, N));
101 //
102 // The most efficient approach is to use emplace(), which
103 // creates just one object "emplace":
104 //
105 this->MapNodes.emplace(id, Node(id, latitude, longitude, entrance));
106
107 //
108 // next node element in the XML doc:
109 //
110 node = node->NextSiblingElement("node");
111 }
112 }
113
114 //
115 // find
116 //
117 // Searches the nodes for the one with the matching ID, returning
118 // true if found and false if not. If found, a copy of the node
119 // is returned via the node parameter, which is passed by reference.
120 //
121 bool Nodes::find(long long id, double& lat, double& lon, bool& isEntrance) const
122 {
123     auto ptr = this->MapNodes.find(id);
124
125     if (ptr == this->MapNodes.end()) { // not found:
126         return false;
127     }
128     else { // found:
129         lat = ptr->second.getLat();
130         lon = ptr->second.getLon();
131         isEntrance = ptr->second.getIsEntrance();
132
133         return true;
134     }
135 }
136
137 //
138 // accessors / getters
139 //
140 int Nodes::getNumMapNodes() {
141     return (int) this->MapNodes.size();
142 }
143
```

```
1  /*nodes.h*/
2
3  //
4  // A collection of nodes in the Open Street Map.
5  //
6  // Prof. Joe Hummel
7  // Northwestern University
8  // CS 211
9  //
10 // References:
11 //
12 // TinyXML:
13 // files: https://github.com/leethomason/tinyxml2
14 // docs: http://leethomason.github.io/tinyxml2/
15 //
16 // OpenStreetMap: https://www.openstreetmap.org
17 // OpenStreetMap docs:
18 // https://wiki.openstreetmap.org/wiki/Main\_Page
19 // https://wiki.openstreetmap.org/wiki/Map\_Features
20 // https://wiki.openstreetmap.org/wiki/Node
21 // https://wiki.openstreetmap.org/wiki/Way
22 // https://wiki.openstreetmap.org/wiki/Relation
23 //
24
25 #pragma once
26
27 #include <map>
28
29 #include "node.h"
30 #include "tinyxml2.h"
31
32 using namespace std;
33 using namespace tinyxml2;
34
35
36 //
37 // Keeps track of all the nodes in the map.
38 //
39 class Nodes
40 {
41 private:
42     map<long long, Node> MapNodes;
43
44 public:
45     //
46     // readMapNodes
```



```
47 //
48 // Given an XML document, reads through the document and
49 // stores all the nodes into the given vector. Each node
50 // is a point on the map, with a unique id along with
51 // (lat, lon) position. Some nodes are entrances to buildings,
52 // which we capture as well.
53 //
54 void readMapNodes(XMLDocument& xmlDoc);
55
56 //
57 // find
58 //
59 // Searches the nodes for the one with the matching ID, returning
60 // true if found and false if not. If found, the node's Lat, Lon,
61 // and IsEntrance data are returned via the reference parameters.
62 //
63 bool find(long long id, double& lat, double& lon, bool& isEntrance) const;
64
65 //
66 // accessors / getters
67 //
68 int getNumMapNodes();
69
70 };
71
72
```

```
1  /*osm.cpp*/
2
3  //
4  // Functions for working with an Open Street Map file.
5  //
6  // Prof. Joe Hummel
7  // Northwestern University
8  // CS 211
9  //
10 // References:
11 //
12 // TinyXML:
13 // files: https://github.com/leethomason/tinyxml2
14 // docs: http://leethomason.github.io/tinyxml2/
15 //
16 // OpenStreetMap: https://www.openstreetmap.org
17 // OpenStreetMap docs:
18 // https://wiki.openstreetmap.org/wiki/Main\_Page
19 // https://wiki.openstreetmap.org/wiki/Map\_Features
20 // https://wiki.openstreetmap.org/wiki/Node
21 // https://wiki.openstreetmap.org/wiki/Way
22 // https://wiki.openstreetmap.org/wiki/Relation
23 //
24
25 #include <iostream>
26 #include <string>
27 #include <cassert>
28
29 #include "osm.h"
30
31 using namespace std;
32 using namespace tinyxml2;
33
34
35 //
36 // osmLoadMapFile
37 //
38 // Given the filename for an XML doc, tries to open and load
39 // that file into the given xmldoc variable (which is passed
40 // by reference). Returns true if successful, false if the
41 // file could not be opened OR the file does not contain
42 // an Open Street Map document.
43 //
44 bool osmLoadMapFile(string filename, XMLDocument& xmldoc)
45 {
46     //
```

```

47 // load the XML document:
48 //
49 xmlDoc.LoadFile(filename.c_str());
50
51 if (xmlDoc.ErrorID() != 0) // failed:
52 {
53     cout << "***ERROR: unable to open XML file '" << filename << "'." << endl;
54     return false;
55 }
56
57 //
58 // top-level element should be "osm" if the file is a valid open
59 // street map:
60 //
61 XMLElement* osm = xmlDoc.FirstChildElement("osm");
62
63 if (osm == nullptr)
64 {
65     cout << "***ERROR: unable to find top-level 'osm' XML element." << endl;
66     cout << "***ERROR: this file is probably not an Open Street Map." << endl;
67     return false;
68 }
69
70 //
71 // success:
72 //
73 return true;
74 }
75
76
77 //
78 // osmContainsKeyValue
79 //
80 // Given a pointer to an XML Element, searches through all
81 // the tags associated with this element looking for the
82 // given (key, value) pair. For example, the call
83 //
84 // containsKeyValue(e, "entrance", "yes")
85 //
86 // will return true if it comes across the tag
87 //
88 // <tag k="entrance" v="yes"/>
89 //
90 bool osmContainsKeyValue(XMLElement* e, string key, string value)
91 {
92     XMLElement* tag = e->FirstChildElement("tag");
93
94     while (tag != nullptr)
95     {

```

```
96     const XMLAttribute* keyAttribute = tag->FindAttribute("k");
97     const XMLAttribute* valueAttribute = tag->FindAttribute("v");
98
99     if (keyAttribute != nullptr && valueAttribute != nullptr)
100     {
101         string elemkey(keyAttribute->Value());
102         string elemvalue(valueAttribute->Value());
103
104         if (elemkey == key && elemvalue == value) // found it:
105         {
106             return true;
107         }
108     }
109
110     //
111     // not a match, try the next tag:
112     //
113     tag = tag->NextSiblingElement("tag");
114 }
115
116 //
117 // if get here, not found:
118 //
119 return false;
120 }
121
122
123 //
124 // osmGetKeyValue
125 //
126 // Given a pointer to an XML Element, searches through all
127 // the tags associated with this element looking for the
128 // given key. If found, returns the associated value. For
129 // example, given the call
130 //
131 //  getKeyValue(e, "entrance")
132 //
133 // will return "yes" if it comes across the tag
134 //
135 //  <tag k="entrance" v="yes"/>
136 //
137 // If the key is not found, the empty string "" is returned.
138 //
139 string osmGetKeyValue(XMLElement* e, string key)
140 {
141     XMLElement* tag = e->FirstChildElement("tag");
142
143     while (tag != nullptr)
144     {
```

```
145     const XMLAttribute* keyAttribute = tag->FindAttribute("k");
146     const XMLAttribute* valueAttribute = tag->FindAttribute("v");
147
148     if (keyAttribute != nullptr && valueAttribute != nullptr)
149     {
150         string elemkey(keyAttribute->Value());
151
152         if (elemkey == key) // found it:
153         {
154             string elemvalue(valueAttribute->Value());
155
156             return elemvalue;
157         }
158     }
159
160     //
161     // not a match, try the next tag:
162     //
163     tag = tag->NextSiblingElement("tag");
164 }
165
166 //
167 // if get here, not found:
168 //
169 return "";
170 }
171
```

```
1  /*osm.h*/
2
3  //
4  // Functions for working with an Open Street Map file.
5  //
6  // Prof. Joe Hummel
7  // Northwestern University
8  // CS 211
9  //
10 // References:
11 //
12 // TinyXML:
13 // files: https://github.com/leethomason/tinyxml2
14 // docs: http://leethomason.github.io/tinyxml2/
15 //
16 // OpenStreetMap: https://www.openstreetmap.org
17 // OpenStreetMap docs:
18 // https://wiki.openstreetmap.org/wiki/Main\_Page
19 // https://wiki.openstreetmap.org/wiki/Map\_Features
20 // https://wiki.openstreetmap.org/wiki/Node
21 // https://wiki.openstreetmap.org/wiki/Way
22 // https://wiki.openstreetmap.org/wiki/Relation
23 //
24
25 #pragma once
26
27 #include "tinyxml2.h"
28
29 using namespace std;
30 using namespace tinyxml2;
31
32
33 //
34 // Helper functions:
35 //
36 bool osmLoadMapFile(string filename, XMLDocument& xmldoc);
37 bool osmContainsKeyValue(XMLElement* e, string key, string value);
38 string osmGetKeyValue(XMLElement* e, string key);
39
```

```
1  /*
2  Original code by Lee Thomason (www.grinninglizard.com)
3
4  This software is provided 'as-is', without any express or implied
5  warranty. In no event will the authors be held liable for any
6  damages arising from the use of this software.
7
8  Permission is granted to anyone to use this software for any
9  purpose, including commercial applications, and to alter it and
10 redistribute it freely, subject to the following restrictions:
11
12 1. The origin of this software must not be misrepresented; you must
13 not claim that you wrote the original software. If you use this
14 software in a product, an acknowledgment in the product documentation
15 would be appreciated but is not required.
16
17 2. Altered source versions must be plainly marked as such, and
18 must not be misrepresented as being the original software.
19
20 3. This notice may not be removed or altered from any source
21 distribution.
22 */
23
24 #include "tinyxml2.h"
25
26 #include <new>          // yes, this one new style header, is in the Android SDK.
27 #if defined(ANDROID_NDK) || defined(__BORLANDC__) || defined(__QNXNTO__)
28 # include <stddef.h>
29 # include <stdarg.h>
30 #else
31 # include <cstddef>
32 # include <cstdarg>
33 #endif
34
35 #if defined(_MSC_VER) && (_MSC_VER >= 1400) && (!defined WINCE)
36     // Microsoft Visual Studio, version 2005 and higher. Not WinCE.
37     /*int _snprintf_s(
38         char *buffer,
39         size_t sizeOfBuffer,
40         size_t count,
41         const char *format [,
42             argument] ...
43     );*/
44     static inline int TIXML_SNPRINTF( char* buffer, size_t size, const char* format, ... )
45     {
46         va_list va;
```

```

47     va_start( va, format );
48     const int result = vsnprintf_s( buffer, size, _TRUNCATE, format, va );
49     va_end( va );
50     return result;
51 }
52
53 static inline int TIXML_VSNPRINTF( char* buffer, size_t size, const char* format, va_list va )
54 {
55     const int result = vsnprintf_s( buffer, size, _TRUNCATE, format, va );
56     return result;
57 }
58
59 #define TIXML_VSCPRINTF    _vscprintf
60 #define TIXML_SSCANF sscanf_s
61 #elif defined _MSC_VER
62     // Microsoft Visual Studio 2003 and earlier or WinCE
63     #define TIXML_SNPRINTF    _snprintf
64     #define TIXML_VSNPRINTF _vsnprintf
65     #define TIXML_SSCANF sscanf
66     #if ( _MSC_VER < 1400 ) && (!defined WINCE)
67         // Microsoft Visual Studio 2003 and not WinCE.
68         #define TIXML_VSCPRINTF _vscprintf // VS2003's C runtime has this, but VC6 C runtime or
WinCE SDK doesn't have.
69     #else
70         // Microsoft Visual Studio 2003 and earlier or WinCE.
71         static inline int TIXML_VSCPRINTF( const char* format, va_list va )
72         {
73             int len = 512;
74             for (;;) {
75                 len = len*2;
76                 char* str = new char[len]();
77                 const int required = _vsnprintf(str, len, format, va);
78                 delete[] str;
79                 if ( required != -1 ) {
80                     TIXMLASSERT( required >= 0 );
81                     len = required;
82                     break;
83                 }
84             }
85             TIXMLASSERT( len >= 0 );
86             return len;
87         }
88     #endif
89 #else
90     // GCC version 3 and higher
91     // #warning( "Using sn* functions." )
92     #define TIXML_SNPRINTF    snprintf
93     #define TIXML_VSNPRINTF    vsnprintf
94     static inline int TIXML_VSCPRINTF( const char* format, va_list va )

```



```

95     {
96         int len = vsnprintf( 0, 0, format, va );
97         TIXMLASSERT( len >= 0 );
98         return len;
99     }
100     #define TIXML_SSCANF  sscanf
101 #endif
102
103 #if defined(_WIN64)
104     #define TIXML_FSEEK _fseeki64
105     #define TIXML_FTELL _ftelli64
106 #elif defined(__APPLE__) || defined(__FreeBSD__) || defined(__OpenBSD__) || defined(__NetBSD__) ||
defined(__DragonFly__) || (__CYGWIN__)
107     #define TIXML_FSEEK fseeko
108     #define TIXML_FTELL ftello
109 #elif defined(__ANDROID__)
110     #if __ANDROID_API__ > 24
111         #define TIXML_FSEEK fseeko64
112         #define TIXML_FTELL ftello64
113     #else
114         #define TIXML_FSEEK fseeko
115         #define TIXML_FTELL ftello
116     #endif
117 #elif defined(__unix__) && defined(__x86_64__)
118     #define TIXML_FSEEK fseeko64
119     #define TIXML_FTELL ftello64
120 #else
121     #define TIXML_FSEEK fseek
122     #define TIXML_FTELL ftell
123 #endif
124
125
126 static const char LINE_FEED                = static_cast<char>(0x0a);           // all line endings are
normalized to LF
127 static const char LF = LINE_FEED;
128 static const char CARRIAGE_RETURN          = static_cast<char>(0x0d);           // CR gets filtered out
129 static const char CR = CARRIAGE_RETURN;
130 static const char SINGLE_QUOTE             = '\'';
131 static const char DOUBLE_QUOTE             = '\"';
132
133 // Bunch of unicode info at:
134 //     http://www.unicode.org/faq/utf_bom.html
135 //     ef bb bf (Microsoft "lead bytes") - designates UTF-8
136
137 static const unsigned char TIXML_UTF_LEAD_0 = 0xefU;
138 static const unsigned char TIXML_UTF_LEAD_1 = 0xbbU;
139 static const unsigned char TIXML_UTF_LEAD_2 = 0xbfU;
140
141 namespace tinyxml2

```

```

142 {
143
144 struct Entity {
145     const char* pattern;
146     int length;
147     char value;
148 };
149
150 static const int NUM_ENTITIES = 5;
151 static const Entity entities[NUM_ENTITIES] = {
152     { "quot", 4, DOUBLE_QUOTE
153 },
154     { "amp", 3,      '&' },
155     { "apos", 4, SINGLE_QUOTE },
156     { "lt", 2,      '<' },
157     { "gt", 2,      '>' }
158 };
159
160 StrPair::~StrPair()
161 {
162     Reset();
163 }
164
165
166 void StrPair::TransferTo( StrPair* other )
167 {
168     if ( this == other ) {
169         return;
170     }
171     // This in effect implements the assignment operator by "moving"
172     // ownership (as in auto_ptr).
173
174     TIXMLASSERT( other != 0 );
175     TIXMLASSERT( other->_flags == 0 );
176     TIXMLASSERT( other->_start == 0 );
177     TIXMLASSERT( other->_end == 0 );
178
179     other->Reset();
180
181     other->_flags = _flags;
182     other->_start = _start;
183     other->_end = _end;
184
185     _flags = 0;
186     _start = 0;
187     _end = 0;
188 }
189

```

```

190
191 void StrPair::Reset()
192 {
193     if ( _flags & NEEDS_DELETE ) {
194         delete [] _start;
195     }
196     _flags = 0;
197     _start = 0;
198     _end = 0;
199 }
200
201
202 void StrPair::SetStr( const char* str, int flags )
203 {
204     TIXMLASSERT( str );
205     Reset();
206     size_t len = strlen( str );
207     TIXMLASSERT( _start == 0 );
208     _start = new char[ len+1 ];
209     memcpy( _start, str, len+1 );
210     _end = _start + len;
211     _flags = flags | NEEDS_DELETE;
212 }
213
214
215 char* StrPair::ParseText( char* p, const char* endTag, int strFlags, int* curLineNumPtr )
216 {
217     TIXMLASSERT( p );
218     TIXMLASSERT( endTag && *endTag );
219     TIXMLASSERT( curLineNumPtr );
220
221     char* start = p;
222     const char endChar = *endTag;
223     size_t length = strlen( endTag );
224
225     // Inner loop of text parsing.
226     while ( *p ) {
227         if ( *p == endChar && strncmp( p, endTag, length ) == 0 ) {
228             Set( start, p, strFlags );
229             return p + length;
230         } else if (*p == '\n') {
231             ++(*curLineNumPtr);
232         }
233         ++p;
234         TIXMLASSERT( p );
235     }
236     return 0;
237 }
238

```

```

239
240 char* StrPair::ParseName( char* p )
241 {
242     if ( !p || !(*p) ) {
243         return 0;
244     }
245     if ( !XMLUtil::IsNameStartChar( (unsigned char) *p ) ) {
246         return 0;
247     }
248
249     char* const start = p;
250     ++p;
251     while ( *p && XMLUtil::IsNameChar( (unsigned char) *p ) ) {
252         ++p;
253     }
254
255     Set( start, p, 0 );
256     return p;
257 }
258
259
260 void StrPair::CollapseWhitespace()
261 {
262     // Adjusting _start would cause undefined behavior on delete[]
263     TIXMLASSERT( ( _flags & NEEDS_DELETE ) == 0 );
264     // Trim leading space.
265     _start = XMLUtil::SkipWhiteSpace( _start, 0 );
266
267     if ( *_start ) {
268         const char* p = _start; // the read pointer
269         char* q = _start; // the write pointer
270
271         while( *p ) {
272             if ( XMLUtil::IsWhiteSpace( *p ) ) {
273                 p = XMLUtil::SkipWhiteSpace( p, 0 );
274                 if ( *p == 0 ) {
275                     break; // don't write to q; this trims the trailing space.
276                 }
277                 *q = ' ';
278                 ++q;
279             }
280             *q = *p;
281             ++q;
282             ++p;
283         }
284         *q = 0;
285     }
286 }
287

```

```

288
289 const char* StrPair::GetStr()
290 {
291     TIXMLASSERT( _start );
292     TIXMLASSERT( _end );
293     if ( _flags & NEEDS_FLUSH ) {
294         *_end = 0;
295         _flags ^= NEEDS_FLUSH;
296
297         if ( _flags ) {
298             const char* p = _start;    // the read pointer
299             char* q = _start; // the write pointer
300
301             while( p < _end ) {
302                 if ( ( _flags & NEEDS_NEWLINE_NORMALIZATION ) && *p == CR ) {
303                     // CR-LF pair becomes LF
304                     // CR alone becomes LF
305                     // LF-CR becomes LF
306                     if ( *(p+1) == LF ) {
307                         p += 2;
308                     }
309                     else {
310                         ++p;
311                     }
312                     *q = LF;
313                     ++q;
314                 }
315                 else if ( ( _flags & NEEDS_NEWLINE_NORMALIZATION ) && *p == LF ) {
316                     if ( *(p+1) == CR ) {
317                         p += 2;
318                     }
319                     else {
320                         ++p;
321                     }
322                     *q = LF;
323                     ++q;
324                 }
325                 else if ( ( _flags & NEEDS_ENTITY_PROCESSING ) && *p == '&' ) {
326                     // Entities handled by tinyXML2:
327                     // - special entities in the entity table [in/out]
328                     // - numeric character reference [in]
329                     //   &#20013; or &#x4e2d;
330
331                     if ( *(p+1) == '#' ) {
332                         const int buflen = 10;
333                         char buf[buflen] = { 0 };
334                         int len = 0;
335                         const char* adjusted = const_cast<char*>( XMLUtil::GetCharacterRef( p, buf, &len ) );
336                         if ( adjusted == 0 ) {

```

```

337         *q = *p;
338         ++p;
339         ++q;
340     }
341     else {
342         TIXMLASSERT( 0 <= len && len <= buflen );
343         TIXMLASSERT( q + len <= adjusted );
344         p = adjusted;
345         memcpy( q, buf, len );
346         q += len;
347     }
348 }
349 else {
350     bool entityFound = false;
351     for( int i = 0; i < NUM_ENTITIES; ++i ) {
352         const Entity& entity = entities[i];
353         if ( strcmp( p + 1, entity.pattern, entity.length ) == 0
354             && *( p + entity.length + 1 ) == ';' ) {
355             // Found an entity - convert.
356             *q = entity.value;
357             ++q;
358             p += entity.length + 2;
359             entityFound = true;
360             break;
361         }
362     }
363     if ( !entityFound ) {
364         // fixme: treat as error?
365         ++p;
366         ++q;
367     }
368 }
369 }
370 else {
371     *q = *p;
372     ++p;
373     ++q;
374 }
375 }
376 *q = 0;
377 }
378 // The loop below has plenty going on, and this
379 // is a less useful mode. Break it out.
380 if ( _flags & NEEDS_WHITESPACE_COLLAPSING ) {
381     CollapseWhitespace();
382 }
383 _flags = ( _flags & NEEDS_DELETE );
384 }
385 TIXMLASSERT( _start );

```

```

386     return _start;
387 }
388
389
390
391
392 // ----- XMLUtil ----- //
393
394 const char* XMLUtil::writeBoolTrue = "true";
395 const char* XMLUtil::writeBoolFalse = "false";
396
397 void XMLUtil::SetBoolSerialization(const char* writeTrue, const char* writeFalse)
398 {
399     static const char* defTrue = "true";
400     static const char* defFalse = "false";
401
402     writeBoolTrue = (writeTrue) ? writeTrue : defTrue;
403     writeBoolFalse = (writeFalse) ? writeFalse : defFalse;
404 }
405
406
407 const char* XMLUtil::ReadBOM( const char* p, bool* bom )
408 {
409     TIXMLASSERT( p );
410     TIXMLASSERT( bom );
411     *bom = false;
412     const unsigned char* pu = reinterpret_cast<const unsigned char*>(p);
413     // Check for BOM:
414     if ( *(pu+0) == TIXML_UTF_LEAD_0
415         && *(pu+1) == TIXML_UTF_LEAD_1
416         && *(pu+2) == TIXML_UTF_LEAD_2 ) {
417         *bom = true;
418         p += 3;
419     }
420     TIXMLASSERT( p );
421     return p;
422 }
423
424
425 void XMLUtil::ConvertUTF32ToUTF8( unsigned long input, char* output, int* length )
426 {
427     const unsigned long BYTE_MASK = 0xBF;
428     const unsigned long BYTE_MARK = 0x80;
429     const unsigned long FIRST_BYTE_MARK[7] = { 0x00, 0x00, 0xC0, 0xE0, 0xF0, 0xF8, 0xFC };
430
431     if (input < 0x80) {
432         *length = 1;
433     }
434     else if ( input < 0x800 ) {

```

```

435     *length = 2;
436 }
437 else if ( input < 0x10000 ) {
438     *length = 3;
439 }
440 else if ( input < 0x200000 ) {
441     *length = 4;
442 }
443 else {
444     *length = 0; // This code won't convert this correctly anyway.
445     return;
446 }
447
448 output += *length;
449
450 // Scary scary fall throughs are annotated with carefully designed comments
451 // to suppress compiler warnings such as -Wimplicit-fallthrough in gcc
452 switch (*length) {
453     case 4:
454         --output;
455         *output = static_cast<char>((input | BYTE_MARK) & BYTE_MASK);
456         input >>= 6;
457         //fall through
458     case 3:
459         --output;
460         *output = static_cast<char>((input | BYTE_MARK) & BYTE_MASK);
461         input >>= 6;
462         //fall through
463     case 2:
464         --output;
465         *output = static_cast<char>((input | BYTE_MARK) & BYTE_MASK);
466         input >>= 6;
467         //fall through
468     case 1:
469         --output;
470         *output = static_cast<char>(input | FIRST_BYTE_MARK[*length]);
471         break;
472     default:
473         TIXMLASSERT( false );
474 }
475 }
476
477
478 const char* XMLUtil::GetCharacterRef( const char* p, char* value, int* length )
479 {
480     // Presume an entity, and pull it out.
481     *length = 0;
482
483     if ( *(p+1) == '#' && *(p+2) ) {

```



```

484 unsigned long ucs = 0;
485 TIXMLASSERT( sizeof( ucs ) >= 4 );
486 ptrdiff_t delta = 0;
487 unsigned mult = 1;
488 static const char SEMICOLON = ';';
489
490 if ( *(p+2) == 'x' ) {
491     // Hexadecimal.
492     const char* q = p+3;
493     if ( !(*q) ) {
494         return 0;
495     }
496
497     q = strchr( q, SEMICOLON );
498
499     if ( !q ) {
500         return 0;
501     }
502     TIXMLASSERT( *q == SEMICOLON );
503
504     delta = q-p;
505     --q;
506
507     while ( *q != 'x' ) {
508         unsigned int digit = 0;
509
510         if ( *q >= '0' && *q <= '9' ) {
511             digit = *q - '0';
512         }
513         else if ( *q >= 'a' && *q <= 'f' ) {
514             digit = *q - 'a' + 10;
515         }
516         else if ( *q >= 'A' && *q <= 'F' ) {
517             digit = *q - 'A' + 10;
518         }
519         else {
520             return 0;
521         }
522         TIXMLASSERT( digit < 16 );
523         TIXMLASSERT( digit == 0 || mult <= UINT_MAX / digit );
524         const unsigned int digitScaled = mult * digit;
525         TIXMLASSERT( ucs <= ULONG_MAX - digitScaled );
526         ucs += digitScaled;
527         TIXMLASSERT( mult <= UINT_MAX / 16 );
528         mult *= 16;
529         --q;
530     }
531 }
532 else {

```

```

533     // Decimal.
534     const char* q = p+2;
535     if ( !(*q) ) {
536         return 0;
537     }
538
539     q = strchr( q, SEMICOLON );
540
541     if ( !q ) {
542         return 0;
543     }
544     TIXMLASSERT( *q == SEMICOLON );
545
546     delta = q-p;
547     --q;
548
549     while ( *q != '#' ) {
550         if ( *q >= '0' && *q <= '9' ) {
551             const unsigned int digit = *q - '0';
552             TIXMLASSERT( digit < 10 );
553             TIXMLASSERT( digit == 0 || mult <= UINT_MAX / digit );
554             const unsigned int digitScaled = mult * digit;
555             TIXMLASSERT( ucs <= ULONG_MAX - digitScaled );
556             ucs += digitScaled;
557         }
558         else {
559             return 0;
560         }
561         TIXMLASSERT( mult <= UINT_MAX / 10 );
562         mult *= 10;
563         --q;
564     }
565 }
566 // convert the UCS to UTF-8
567 ConvertUTF32ToUTF8( ucs, value, length );
568 return p + delta + 1;
569 }
570 return p+1;
571 }
572
573
574 void XMLUtil::ToStr( int v, char* buffer, int bufferSize )
575 {
576     TIXML_SNPRINTF( buffer, bufferSize, "%d", v );
577 }
578
579
580 void XMLUtil::ToStr( unsigned v, char* buffer, int bufferSize )
581 {

```

```

582     TIXML_SNPRINTF( buffer, bufferSize, "%u", v );
583 }
584
585
586 void XMLUtil::ToStr( bool v, char* buffer, int bufferSize )
587 {
588     TIXML_SNPRINTF( buffer, bufferSize, "%s", v ? writeBoolTrue : writeBoolFalse);
589 }
590
591 /*
592     ToStr() of a number is a very tricky topic.
593     https://github.com/leethomason/tinyxml2/issues/106
594 */
595 void XMLUtil::ToStr( float v, char* buffer, int bufferSize )
596 {
597     TIXML_SNPRINTF( buffer, bufferSize, "%.8g", v );
598 }
599
600
601 void XMLUtil::ToStr( double v, char* buffer, int bufferSize )
602 {
603     TIXML_SNPRINTF( buffer, bufferSize, "%.17g", v );
604 }
605
606
607 void XMLUtil::ToStr( int64_t v, char* buffer, int bufferSize )
608 {
609     // horrible syntax trick to make the compiler happy about %lld
610     TIXML_SNPRINTF(buffer, bufferSize, "%lld", static_cast<long long>(v));
611 }
612
613 void XMLUtil::ToStr( uint64_t v, char* buffer, int bufferSize )
614 {
615     // horrible syntax trick to make the compiler happy about %llu
616     TIXML_SNPRINTF(buffer, bufferSize, "%llu", (long long)v);
617 }
618
619 bool XMLUtil::ToInt(const char* str, int* value)
620 {
621     if (IsPrefixHex(str)) {
622         unsigned v;
623         if (TIXML_SSCANF(str, "%x", &v) == 1) {
624             *value = static_cast<int>(v);
625             return true;
626         }
627     }
628     else {
629         if (TIXML_SSCANF(str, "%d", value) == 1) {
630             return true;

```

```
631     }
632 }
633 return false;
634 }
635
636 bool XMLUtil::ToUnsigned(const char* str, unsigned* value)
637 {
638     if (TIXML_SSCANF(str, IsPrefixHex(str) ? "%x" : "%u", value) == 1) {
639         return true;
640     }
641     return false;
642 }
643
644 bool XMLUtil::ToBool( const char* str, bool* value )
645 {
646     int ival = 0;
647     if ( ToInt( str, &ival )) {
648         *value = (ival==0) ? false : true;
649         return true;
650     }
651     static const char* TRUE_VALS[] = { "true", "True", "TRUE", 0 };
652     static const char* FALSE_VALS[] = { "false", "False", "FALSE", 0 };
653
654     for (int i = 0; TRUE_VALS[i]; ++i) {
655         if (StringEqual(str, TRUE_VALS[i])) {
656             *value = true;
657             return true;
658         }
659     }
660     for (int i = 0; FALSE_VALS[i]; ++i) {
661         if (StringEqual(str, FALSE_VALS[i])) {
662             *value = false;
663             return true;
664         }
665     }
666     return false;
667 }
668
669
670 bool XMLUtil::ToFloat( const char* str, float* value )
671 {
672     if ( TIXML_SSCANF( str, "%f", value ) == 1 ) {
673         return true;
674     }
675     return false;
676 }
677
678
679 bool XMLUtil::ToDouble( const char* str, double* value )
```

```

680 {
681     if ( TIXML_SSCANF( str, "%lf", value ) == 1 ) {
682         return true;
683     }
684     return false;
685 }
686
687
688 bool XMLUtil::ToInt64(const char* str, int64_t* value)
689 {
690     if (IsPrefixHex(str)) {
691         unsigned long long v = 0;    // horrible syntax trick to make the compiler happy about %llx
692         if (TIXML_SSCANF(str, "%llx", &v) == 1) {
693             *value = static_cast<int64_t>(v);
694             return true;
695         }
696     }
697     else {
698         long long v = 0;    // horrible syntax trick to make the compiler happy about %lld
699         if (TIXML_SSCANF(str, "%lld", &v) == 1) {
700             *value = static_cast<int64_t>(v);
701             return true;
702         }
703     }
704     return false;
705 }
706
707
708 bool XMLUtil::ToUnsigned64(const char* str, uint64_t* value) {
709     unsigned long long v = 0; // horrible syntax trick to make the compiler happy about %llu
710     if(TIXML_SSCANF(str, IsPrefixHex(str) ? "%llx" : "%llu", &v) == 1) {
711         *value = (uint64_t)v;
712         return true;
713     }
714     return false;
715 }
716
717
718 char* XMLDocument::Identify( char* p, XMLNode** node )
719 {
720     TIXMLASSERT( node );
721     TIXMLASSERT( p );
722     char* const start = p;
723     int const startLine = _parseCurLineNum;
724     p = XMLUtil::SkipWhiteSpace( p, &_parseCurLineNum );
725     if( !*p ) {
726         *node = 0;
727         TIXMLASSERT( p );
728         return p;

```

```

729 }
730
731 // These strings define the matching patterns:
732 static const char* xmlHeader      = { "<?" };
733 static const char* commentHeader = { "<!--" };
734 static const char* cdataHeader   = { "<![CDATA[" };
735 static const char* dtdHeader     = { "<!" };
736 static const char* elementHeader = { "<" }; // and a header for everything else; check last.
737
738 static const int xmlHeaderLen      = 2;
739 static const int commentHeaderLen = 4;
740 static const int cdataHeaderLen   = 9;
741 static const int dtdHeaderLen     = 2;
742 static const int elementHeaderLen = 1;
743
744 TIXMLASSERT( sizeof( XMLComment ) == sizeof( XMLUnknown ) ); // use same memory pool
745 TIXMLASSERT( sizeof( XMLComment ) == sizeof( XMLDeclaration ) ); // use same memory pool
746 XMLNode* returnNode = 0;
747 if ( XMLUtil::StringEqual( p, xmlHeader, xmlHeaderLen ) ) {
748     returnNode = CreateUnlinkedNode<XMLDeclaration>( _commentPool );
749     returnNode->_parseLineNum = _parseCurLineNum;
750     p += xmlHeaderLen;
751 }
752 else if ( XMLUtil::StringEqual( p, commentHeader, commentHeaderLen ) ) {
753     returnNode = CreateUnlinkedNode<XMLComment>( _commentPool );
754     returnNode->_parseLineNum = _parseCurLineNum;
755     p += commentHeaderLen;
756 }
757 else if ( XMLUtil::StringEqual( p, cdataHeader, cdataHeaderLen ) ) {
758     XMLText* text = CreateUnlinkedNode<XMLText>( _textPool );
759     returnNode = text;
760     returnNode->_parseLineNum = _parseCurLineNum;
761     p += cdataHeaderLen;
762     text->SetCData( true );
763 }
764 else if ( XMLUtil::StringEqual( p, dtdHeader, dtdHeaderLen ) ) {
765     returnNode = CreateUnlinkedNode<XMLUnknown>( _commentPool );
766     returnNode->_parseLineNum = _parseCurLineNum;
767     p += dtdHeaderLen;
768 }
769 else if ( XMLUtil::StringEqual( p, elementHeader, elementHeaderLen ) ) {
770     returnNode = CreateUnlinkedNode<XMLElement>( _elementPool );
771     returnNode->_parseLineNum = _parseCurLineNum;
772     p += elementHeaderLen;
773 }
774 else {
775     returnNode = CreateUnlinkedNode<XMLText>( _textPool );
776     returnNode->_parseLineNum = _parseCurLineNum; // Report line of first non-whitespace
character

```

```

777     p = start; // Back it up, all the text counts.
778     _parseCurLineNum = startLine;
779 }
780
781 TIXMLASSERT( returnNode );
782 TIXMLASSERT( p );
783 *node = returnNode;
784 return p;
785 }
786
787
788 bool XMLDocument::Accept( XMLVisitor* visitor ) const
789 {
790     TIXMLASSERT( visitor );
791     if ( visitor->VisitEnter( *this ) ) {
792         for ( const XMLNode* node=FirstChild(); node; node=node->NextSibling() ) {
793             if ( !node->Accept( visitor ) ) {
794                 break;
795             }
796         }
797     }
798     return visitor->VisitExit( *this );
799 }
800
801
802 // ----- XMLNode ----- //
803
804 XMLNode::XMLNode( XMLDocument* doc ) :
805     _document( doc ),
806     _parent( 0 ),
807     _value(),
808     _parseLineNum( 0 ),
809     _firstChild( 0 ), _lastChild( 0 ),
810     _prev( 0 ), _next( 0 ),
811     _userData( 0 ),
812     _memPool( 0 )
813 {
814 }
815
816
817 XMLNode::~XMLNode()
818 {
819     DeleteChildren();
820     if ( _parent ) {
821         _parent->Unlink( this );
822     }
823 }
824
825 const char* XMLNode::Value() const

```

```

826 {
827     // Edge case: XMLDocuments don't have a Value. Return null.
828     if ( this->ToDocument() )
829         return 0;
830     return _value.GetStr();
831 }
832
833 void XMLNode::SetValue( const char* str, bool staticMem )
834 {
835     if ( staticMem ) {
836         _value.SetInternedStr( str );
837     }
838     else {
839         _value.SetStr( str );
840     }
841 }
842
843 XMLNode* XMLNode::DeepClone(XMLDocument* target) const
844 {
845     XMLNode* clone = this->ShallowClone(target);
846     if (!clone) return 0;
847
848     for (const XMLNode* child = this->FirstChild(); child; child = child->NextSibling()) {
849         XMLNode* childClone = child->DeepClone(target);
850         TIXMLASSERT(childClone);
851         clone->InsertEndChild(childClone);
852     }
853     return clone;
854 }
855
856 void XMLNode::DeleteChildren()
857 {
858     while( _firstChild ) {
859         TIXMLASSERT( _lastChild );
860         DeleteChild( _firstChild );
861     }
862     _firstChild = _lastChild = 0;
863 }
864
865
866 void XMLNode::Unlink( XMLNode* child )
867 {
868     TIXMLASSERT( child );
869     TIXMLASSERT( child->_document == _document );
870     TIXMLASSERT( child->_parent == this );
871     if ( child == _firstChild ) {
872         _firstChild = _firstChild->_next;
873     }
874     if ( child == _lastChild ) {

```



```

875     _lastChild = _lastChild->_prev;
876 }
877
878 if ( child->_prev ) {
879     child->_prev->_next = child->_next;
880 }
881 if ( child->_next ) {
882     child->_next->_prev = child->_prev;
883 }
884 child->_next = 0;
885 child->_prev = 0;
886 child->_parent = 0;
887 }
888
889
890 void XMLNode::DeleteChild( XMLNode* node )
891 {
892     TIXMLASSERT( node );
893     TIXMLASSERT( node->_document == _document );
894     TIXMLASSERT( node->_parent == this );
895     Unlink( node );
896     TIXMLASSERT( node->_prev == 0 );
897     TIXMLASSERT( node->_next == 0 );
898     TIXMLASSERT( node->_parent == 0 );
899     DeleteNode( node );
900 }
901
902
903 XMLNode* XMLNode::InsertEndChild( XMLNode* addThis )
904 {
905     TIXMLASSERT( addThis );
906     if ( addThis->_document != _document ) {
907         TIXMLASSERT( false );
908         return 0;
909     }
910     InsertChildPreamble( addThis );
911
912     if ( _lastChild ) {
913         TIXMLASSERT( _firstChild );
914         TIXMLASSERT( _lastChild->_next == 0 );
915         _lastChild->_next = addThis;
916         addThis->_prev = _lastChild;
917         _lastChild = addThis;
918
919         addThis->_next = 0;
920     }
921     else {
922         TIXMLASSERT( _firstChild == 0 );
923         _firstChild = _lastChild = addThis;

```

```
924
925     addThis->_prev = 0;
926     addThis->_next = 0;
927 }
928 addThis->_parent = this;
929 return addThis;
930 }
931
932
933 XMLNode* XMLNode::InsertFirstChild( XMLNode* addThis )
934 {
935     TIXMLASSERT( addThis );
936     if ( addThis->_document != _document ) {
937         TIXMLASSERT( false );
938         return 0;
939     }
940     InsertChildPreamble( addThis );
941
942     if ( _firstChild ) {
943         TIXMLASSERT( _lastChild );
944         TIXMLASSERT( _firstChild->_prev == 0 );
945
946         _firstChild->_prev = addThis;
947         addThis->_next = _firstChild;
948         _firstChild = addThis;
949
950         addThis->_prev = 0;
951     }
952     else {
953         TIXMLASSERT( _lastChild == 0 );
954         _firstChild = _lastChild = addThis;
955
956         addThis->_prev = 0;
957         addThis->_next = 0;
958     }
959     addThis->_parent = this;
960     return addThis;
961 }
962
963
964 XMLNode* XMLNode::InsertAfterChild( XMLNode* afterThis, XMLNode* addThis )
965 {
966     TIXMLASSERT( addThis );
967     if ( addThis->_document != _document ) {
968         TIXMLASSERT( false );
969         return 0;
970     }
971
972     TIXMLASSERT( afterThis );
```

```

973
974 if ( afterThis->_parent != this ) {
975     TIXMLASSERT( false );
976     return 0;
977 }
978 if ( afterThis == addThis ) {
979     // Current state: BeforeThis -> AddThis -> OneAfterAddThis
980     // Now AddThis must disappear from it's location and then
981     // reappear between BeforeThis and OneAfterAddThis.
982     // So just leave it where it is.
983     return addThis;
984 }
985
986 if ( afterThis->_next == 0 ) {
987     // The last node or the only node.
988     return InsertEndChild( addThis );
989 }
990 InsertChildPreamble( addThis );
991 addThis->_prev = afterThis;
992 addThis->_next = afterThis->_next;
993 afterThis->_next->_prev = addThis;
994 afterThis->_next = addThis;
995 addThis->_parent = this;
996 return addThis;
997 }
998
999
1000
1001
1002 const XMLElement* XMLNode::FirstChildElement( const char* name ) const
1003 {
1004     for( const XMLNode* node = _firstChild; node; node = node->_next ) {
1005         const XMLElement* element = node->ToElementWithName( name );
1006         if ( element ) {
1007             return element;
1008         }
1009     }
1010     return 0;
1011 }
1012
1013
1014 const XMLElement* XMLNode::LastChildElement( const char* name ) const
1015 {
1016     for( const XMLNode* node = _lastChild; node; node = node->_prev ) {
1017         const XMLElement* element = node->ToElementWithName( name );
1018         if ( element ) {
1019             return element;
1020         }
1021     }

```

```

1022     return 0;
1023 }
1024
1025
1026 const XMLElement* XMLNode::NextSiblingElement( const char* name ) const
1027 {
1028     for( const XMLNode* node = _next; node; node = node->_next ) {
1029         const XMLElement* element = node->ToElementWithName( name );
1030         if ( element ) {
1031             return element;
1032         }
1033     }
1034     return 0;
1035 }
1036
1037
1038 const XMLElement* XMLNode::PreviousSiblingElement( const char* name ) const
1039 {
1040     for( const XMLNode* node = _prev; node; node = node->_prev ) {
1041         const XMLElement* element = node->ToElementWithName( name );
1042         if ( element ) {
1043             return element;
1044         }
1045     }
1046     return 0;
1047 }
1048
1049
1050 char* XMLNode::ParseDeep( char* p, StrPair* parentEndTag, int* curLineNumPtr )
1051 {
1052     // This is a recursive method, but thinking about it "at the current level"
1053     // it is a pretty simple flat list:
1054     //     <foo/>
1055     //     <!-- comment -->
1056     //
1057     // With a special case:
1058     //     <foo>
1059     //     </foo>
1060     //     <!-- comment -->
1061     //
1062     // Where the closing element (/foo) *must* be the next thing after the opening
1063     // element, and the names must match. BUT the tricky bit is that the closing
1064     // element will be read by the child.
1065     //
1066     // 'endTag' is the end tag for this node, it is returned by a call to a child.
1067     // 'parentEnd' is the end tag for the parent, which is filled in and returned.
1068
1069     XMLDocument::DepthTracker tracker(_document);
1070     if (_document->Error())

```

```

1071         return 0;
1072
1073     while( p && *p ) {
1074         XMLNode* node = 0;
1075
1076         p = _document->Identify( p, &node );
1077         TIXMLASSERT( p );
1078         if ( node == 0 ) {
1079             break;
1080         }
1081
1082         const int initialLineNum = node->_parseLineNum;
1083
1084         StrPair endTag;
1085         p = node->ParseDeep( p, &endTag, curLineNumPtr );
1086         if ( !p ) {
1087             _document->DeleteNode( node );
1088             if ( !_document->Error() ) {
1089                 _document->SetError( XML_ERROR_PARSING, initialLineNum, 0);
1090             }
1091             break;
1092         }
1093
1094         const XMLDeclaration* const decl = node->ToDeclaration();
1095         if ( decl ) {
1096             // Declarations are only allowed at document level
1097             //
1098             // Multiple declarations are allowed but all declarations
1099             // must occur before anything else.
1100             //
1101             // Optimized due to a security test case. If the first node is
1102             // a declaration, and the last node is a declaration, then only
1103             // declarations have so far been added.
1104             bool wellLocated = false;
1105
1106             if (ToDocument()) {
1107                 if (FirstChild()) {
1108                     wellLocated =
1109                         FirstChild() &&
1110                         FirstChild()->ToDeclaration() &&
1111                         LastChild() &&
1112                         LastChild()->ToDeclaration();
1113                 }
1114                 else {
1115                     wellLocated = true;
1116                 }
1117             }
1118             if ( !wellLocated ) {
1119                 _document->SetError( XML_ERROR_PARSING_DECLARATION, initialLineNum, "XMLDeclaration

```

```

        value=%s", decl->Value());
1120     _document->DeleteNode( node );
1121     break;
1122 }
1123 }
1124
1125 XMLElement* ele = node->ToElement();
1126 if ( ele ) {
1127     // We read the end tag. Return it to the parent.
1128     if ( ele->ClosingType() == XMLElement::CLOSING ) {
1129         if ( parentEndTag ) {
1130             ele->_value.TransferTo( parentEndTag );
1131         }
1132         node->_memPool->SetTracked(); // created and then immediately deleted.
1133         DeleteNode( node );
1134         return p;
1135     }
1136
1137     // Handle an end tag returned to this level.
1138     // And handle a bunch of annoying errors.
1139     bool mismatch = false;
1140     if ( endTag.Empty() ) {
1141         if ( ele->ClosingType() == XMLElement::OPEN ) {
1142             mismatch = true;
1143         }
1144     }
1145     else {
1146         if ( ele->ClosingType() != XMLElement::OPEN ) {
1147             mismatch = true;
1148         }
1149         else if ( !XMLUtil::StringEqual( endTag.GetStr(), ele->Name() ) ) {
1150             mismatch = true;
1151         }
1152     }
1153     if ( mismatch ) {
1154         _document->SetError( XML_ERROR_MISMATCHED_ELEMENT, initialLineNum, "XMLElement
name=%s", ele->Name());
1155         _document->DeleteNode( node );
1156         break;
1157     }
1158 }
1159 InsertEndChild( node );
1160 }
1161 return 0;
1162 }
1163
1164 /*static*/ void XMLNode::DeleteNode( XMLNode* node )
1165 {
1166     if ( node == 0 ) {

```

```

1167     return;
1168 }
1169 TIXMLASSERT(node->_document);
1170 if (!node->ToDocument()) {
1171     node->_document->MarkInUse(node);
1172 }
1173
1174 MemPool* pool = node->_memPool;
1175 node->~XMLNode();
1176 pool->Free( node );
1177 }
1178
1179 void XMLNode::InsertChildPreamble( XMLNode* insertThis ) const
1180 {
1181     TIXMLASSERT( insertThis );
1182     TIXMLASSERT( insertThis->_document == _document );
1183
1184     if (insertThis->_parent) {
1185         insertThis->_parent->Unlink( insertThis );
1186     }
1187     else {
1188         insertThis->_document->MarkInUse(insertThis);
1189         insertThis->_memPool->SetTracked();
1190     }
1191 }
1192
1193 const XMLElement* XMLNode::ToElementWithName( const char* name ) const
1194 {
1195     const XMLElement* element = this->ToElement();
1196     if ( element == 0 ) {
1197         return 0;
1198     }
1199     if ( name == 0 ) {
1200         return element;
1201     }
1202     if ( XMLUtil::StringEqual( element->Name(), name ) ) {
1203         return element;
1204     }
1205     return 0;
1206 }
1207
1208 // ----- XMLText ----- //
1209 char* XMLText::ParseDeep( char* p, StrPair*, int* curLineNumPtr )
1210 {
1211     if ( this->CData() ) {
1212         p = _value.ParseText( p, "]]>", StrPair::NEEDS_NEWLINE_NORMALIZATION, curLineNumPtr );
1213         if ( !p ) {
1214             _document->SetError( XML_ERROR_PARSING_CDATA, _parseLineNum, 0 );
1215         }
1216     }
1217 }

```

```

1216     return p;
1217 }
1218 else {
1219     int flags = _document->ProcessEntities() ? StrPair::TEXT_ELEMENT :
StrPair::TEXT_ELEMENT_LEAVE_ENTITIES;
1220     if ( _document->WhitespaceMode() == COLLAPSE_WHITESPACE ) {
1221         flags |= StrPair::NEEDS_WHITESPACE_COLLAPSING;
1222     }
1223
1224     p = _value.ParseText( p, "<", flags, curLineNumPtr );
1225     if ( p && *p ) {
1226         return p-1;
1227     }
1228     if ( !p ) {
1229         _document->SetError( XML_ERROR_PARSING_TEXT, _parseLineNum, 0 );
1230     }
1231 }
1232 return 0;
1233 }
1234
1235
1236 XMLNode* XMLText::ShallowClone( XMLDocument* doc ) const
1237 {
1238     if ( !doc ) {
1239         doc = _document;
1240     }
1241     XMLText* text = doc->NewText( Value() ); // fixme: this will always allocate memory. Intern?
1242     text->SetCData( this->CData() );
1243     return text;
1244 }
1245
1246
1247 bool XMLText::ShallowEqual( const XMLNode* compare ) const
1248 {
1249     TIXMLASSERT( compare );
1250     const XMLText* text = compare->ToText();
1251     return ( text && XMLUtil::StringEqual( text->Value(), Value() ) );
1252 }
1253
1254
1255 bool XMLText::Accept( XMLVisitor* visitor ) const
1256 {
1257     TIXMLASSERT( visitor );
1258     return visitor->Visit( *this );
1259 }
1260
1261
1262 // ----- XMLComment ----- //
1263

```



```

1264 XMLComment::XMLComment( XMLDocument* doc ) : XMLNode( doc )
1265 {
1266 }
1267
1268
1269 XMLComment::~~XMLComment()
1270 {
1271 }
1272
1273
1274 char* XMLComment::ParseDeep( char* p, StrPair*, int* curLineNumPtr )
1275 {
1276     // Comment parses as text.
1277     p = _value.ParseText( p, "-->", StrPair::COMMENT, curLineNumPtr );
1278     if ( p == 0 ) {
1279         _document->SetError( XML_ERROR_PARSING_COMMENT, _parseLineNum, 0 );
1280     }
1281     return p;
1282 }
1283
1284
1285 XMLNode* XMLComment::ShallowClone( XMLDocument* doc ) const
1286 {
1287     if ( !doc ) {
1288         doc = _document;
1289     }
1290     XMLComment* comment = doc->NewComment( Value() );    // fixme: this will always allocate memory.
1291     Intern?
1292     return comment;
1293 }
1294
1295 bool XMLComment::ShallowEqual( const XMLNode* compare ) const
1296 {
1297     TIXMLASSERT( compare );
1298     const XMLComment* comment = compare->ToComment();
1299     return ( comment && XMLUtil::StringEqual( comment->Value(), Value() ) );
1300 }
1301
1302
1303 bool XMLComment::Accept( XMLVisitor* visitor ) const
1304 {
1305     TIXMLASSERT( visitor );
1306     return visitor->Visit( *this );
1307 }
1308
1309
1310 // ----- XMLDeclaration ----- //
1311

```

```

1312 XMLDeclaration::XMLDeclaration( XMLDocument* doc ) : XMLNode( doc )
1313 {
1314 }
1315
1316
1317 XMLDeclaration::~XMLDeclaration()
1318 {
1319     //printf( "~XMLDeclaration\n" );
1320 }
1321
1322
1323 char* XMLDeclaration::ParseDeep( char* p, StrPair*, int* curLineNumPtr )
1324 {
1325     // Declaration parses as text.
1326     p = _value.ParseText( p, ">", StrPair::NEEDS_NEWLINE_NORMALIZATION, curLineNumPtr );
1327     if ( p == 0 ) {
1328         _document->SetError( XML_ERROR_PARSING_DECLARATION, _parseLineNum, 0 );
1329     }
1330     return p;
1331 }
1332
1333
1334 XMLNode* XMLDeclaration::ShallowClone( XMLDocument* doc ) const
1335 {
1336     if ( !doc ) {
1337         doc = _document;
1338     }
1339     XMLDeclaration* dec = doc->NewDeclaration( Value() ); // fixme: this will always allocate memory.
1340     Intern?
1341     return dec;
1342 }
1343
1344 bool XMLDeclaration::ShallowEqual( const XMLNode* compare ) const
1345 {
1346     TIXMLASSERT( compare );
1347     const XMLDeclaration* declaration = compare->ToDeclaration();
1348     return ( declaration && XMLUtil::StringEqual( declaration->Value(), Value() ) );
1349 }
1350
1351
1352
1353 bool XMLDeclaration::Accept( XMLVisitor* visitor ) const
1354 {
1355     TIXMLASSERT( visitor );
1356     return visitor->Visit( *this );
1357 }
1358
1359 // ----- XMLUnknown ----- //

```

```

1360
1361 XMLUnknown::XMLUnknown( XMLDocument* doc ) : XMLNode( doc )
1362 {
1363 }
1364
1365
1366 XMLUnknown::~~XMLUnknown()
1367 {
1368 }
1369
1370
1371 char* XMLUnknown::ParseDeep( char* p, StrPair*, int* curLineNumPtr )
1372 {
1373     // Unknown parses as text.
1374     p = _value.ParseText( p, ">", StrPair::NEEDS_NEWLINE_NORMALIZATION, curLineNumPtr );
1375     if ( !p ) {
1376         _document->SetError( XML_ERROR_PARSING_UNKNOWN, _parseLineNum, 0 );
1377     }
1378     return p;
1379 }
1380
1381
1382 XMLNode* XMLUnknown::ShallowClone( XMLDocument* doc ) const
1383 {
1384     if ( !doc ) {
1385         doc = _document;
1386     }
1387     XMLUnknown* text = doc->NewUnknown( Value() );    // fixme: this will always allocate memory.
1388     Intern?
1389     return text;
1390 }
1391
1392 bool XMLUnknown::ShallowEqual( const XMLNode* compare ) const
1393 {
1394     TIXMLASSERT( compare );
1395     const XMLUnknown* unknown = compare->ToUnknown();
1396     return ( unknown && XMLUtil::StringEqual( unknown->Value(), Value() ) );
1397 }
1398
1399
1400 bool XMLUnknown::Accept( XMLVisitor* visitor ) const
1401 {
1402     TIXMLASSERT( visitor );
1403     return visitor->Visit( *this );
1404 }
1405
1406 // ----- XMLAttribute ----- //
1407

```

```
1408 const char* XMLAttribute::Name() const
1409 {
1410     return _name.GetStr();
1411 }
1412
1413 const char* XMLAttribute::Value() const
1414 {
1415     return _value.GetStr();
1416 }
1417
1418 char* XMLAttribute::ParseDeep( char* p, bool processEntities, int* curLineNumPtr )
1419 {
1420     // Parse using the name rules: bug fix, was using ParseText before
1421     p = _name.ParseName( p );
1422     if ( !p || !*p ) {
1423         return 0;
1424     }
1425
1426     // Skip white space before =
1427     p = XMLUtil::SkipWhiteSpace( p, curLineNumPtr );
1428     if ( *p != '=' ) {
1429         return 0;
1430     }
1431
1432     ++p; // move up to opening quote
1433     p = XMLUtil::SkipWhiteSpace( p, curLineNumPtr );
1434     if ( *p != '"' && *p != '\'' ) {
1435         return 0;
1436     }
1437
1438     const char endTag[2] = { *p, 0 };
1439     ++p; // move past opening quote
1440
1441     p = _value.ParseText( p, endTag, processEntities ? StrPair::ATTRIBUTE_VALUE :
StrPair::ATTRIBUTE_VALUE_LEAVE_ENTITIES, curLineNumPtr );
1442     return p;
1443 }
1444
1445
1446 void XMLAttribute::SetName( const char* n )
1447 {
1448     _name.SetStr( n );
1449 }
1450
1451
1452 XMLError XMLAttribute::QueryIntValue( int* value ) const
1453 {
1454     if ( XMLUtil::ToInt( Value(), value ) ) {
1455         return XML_SUCCESS;
```

```
1456     }
1457     return XML_WRONG_ATTRIBUTE_TYPE;
1458 }
1459
1460
1461 XML_Error XMLAttribute::QueryUnsignedValue( unsigned int* value ) const
1462 {
1463     if ( XMLUtil::ToUnsigned( Value(), value ) ) {
1464         return XML_SUCCESS;
1465     }
1466     return XML_WRONG_ATTRIBUTE_TYPE;
1467 }
1468
1469
1470 XML_Error XMLAttribute::QueryInt64Value(int64_t* value) const
1471 {
1472     if (XMLUtil::ToInt64(Value(), value)) {
1473         return XML_SUCCESS;
1474     }
1475     return XML_WRONG_ATTRIBUTE_TYPE;
1476 }
1477
1478
1479 XML_Error XMLAttribute::QueryUnsigned64Value(uint64_t* value) const
1480 {
1481     if(XMLUtil::ToUnsigned64(Value(), value)) {
1482         return XML_SUCCESS;
1483     }
1484     return XML_WRONG_ATTRIBUTE_TYPE;
1485 }
1486
1487
1488 XML_Error XMLAttribute::QueryBoolValue( bool* value ) const
1489 {
1490     if ( XMLUtil::ToBool( Value(), value ) ) {
1491         return XML_SUCCESS;
1492     }
1493     return XML_WRONG_ATTRIBUTE_TYPE;
1494 }
1495
1496
1497 XML_Error XMLAttribute::QueryFloatValue( float* value ) const
1498 {
1499     if ( XMLUtil::ToFloat( Value(), value ) ) {
1500         return XML_SUCCESS;
1501     }
1502     return XML_WRONG_ATTRIBUTE_TYPE;
1503 }
1504
```

```
1505
1506 XMLError XMLAttribute::QueryDoubleValue( double* value ) const
1507 {
1508     if ( XMLUtil::ToDouble( Value(), value )) {
1509         return XML_SUCCESS;
1510     }
1511     return XML_WRONG_ATTRIBUTE_TYPE;
1512 }
1513
1514
1515 void XMLAttribute::SetAttribute( const char* v )
1516 {
1517     _value.SetStr( v );
1518 }
1519
1520
1521 void XMLAttribute::SetAttribute( int v )
1522 {
1523     char buf[BUF_SIZE];
1524     XMLUtil::ToStr( v, buf, BUF_SIZE );
1525     _value.SetStr( buf );
1526 }
1527
1528
1529 void XMLAttribute::SetAttribute( unsigned v )
1530 {
1531     char buf[BUF_SIZE];
1532     XMLUtil::ToStr( v, buf, BUF_SIZE );
1533     _value.SetStr( buf );
1534 }
1535
1536
1537 void XMLAttribute::SetAttribute(int64_t v)
1538 {
1539     char buf[BUF_SIZE];
1540     XMLUtil::ToStr(v, buf, BUF_SIZE);
1541     _value.SetStr(buf);
1542 }
1543
1544 void XMLAttribute::SetAttribute(uint64_t v)
1545 {
1546     char buf[BUF_SIZE];
1547     XMLUtil::ToStr(v, buf, BUF_SIZE);
1548     _value.SetStr(buf);
1549 }
1550
1551
1552 void XMLAttribute::SetAttribute( bool v )
1553 {
```

```

1554     char buf[BUF_SIZE];
1555     XMLUtil::ToStr( v, buf, BUF_SIZE );
1556     _value.SetStr( buf );
1557 }
1558
1559 void XMLAttribute::SetAttribute( double v )
1560 {
1561     char buf[BUF_SIZE];
1562     XMLUtil::ToStr( v, buf, BUF_SIZE );
1563     _value.SetStr( buf );
1564 }
1565
1566 void XMLAttribute::SetAttribute( float v )
1567 {
1568     char buf[BUF_SIZE];
1569     XMLUtil::ToStr( v, buf, BUF_SIZE );
1570     _value.SetStr( buf );
1571 }
1572
1573
1574 // ----- XMLElement ----- //
1575 XMLElement::XMLElement( XMLDocument* doc ) : XMLNode( doc ),
1576     _closingType( OPEN ),
1577     _rootAttribute( 0 )
1578 {
1579 }
1580
1581
1582 XMLElement::~XMLElement()
1583 {
1584     while( _rootAttribute ) {
1585         XMLAttribute* next = _rootAttribute->_next;
1586         DeleteAttribute( _rootAttribute );
1587         _rootAttribute = next;
1588     }
1589 }
1590
1591
1592 const XMLAttribute* XMLElement::FindAttribute( const char* name ) const
1593 {
1594     for( XMLAttribute* a = _rootAttribute; a; a = a->_next ) {
1595         if ( XMLUtil::StringEqual( a->Name(), name ) ) {
1596             return a;
1597         }
1598     }
1599     return 0;
1600 }
1601
1602

```

```
1603 const char* XMLElement::Attribute( const char* name, const char* value ) const
1604 {
1605     const XMLAttribute* a = FindAttribute( name );
1606     if ( !a ) {
1607         return 0;
1608     }
1609     if ( !value || XMLUtil::StringEqual( a->Value(), value ) ) {
1610         return a->Value();
1611     }
1612     return 0;
1613 }
1614
1615 int XMLElement::IntAttribute(const char* name, int defaultValue) const
1616 {
1617     int i = defaultValue;
1618     QueryIntAttribute(name, &i);
1619     return i;
1620 }
1621
1622 unsigned XMLElement::UnsignedAttribute(const char* name, unsigned defaultValue) const
1623 {
1624     unsigned i = defaultValue;
1625     QueryUnsignedAttribute(name, &i);
1626     return i;
1627 }
1628
1629 int64_t XMLElement::Int64Attribute(const char* name, int64_t defaultValue) const
1630 {
1631     int64_t i = defaultValue;
1632     QueryInt64Attribute(name, &i);
1633     return i;
1634 }
1635
1636 uint64_t XMLElement::Unsigned64Attribute(const char* name, uint64_t defaultValue) const
1637 {
1638     uint64_t i = defaultValue;
1639     QueryUnsigned64Attribute(name, &i);
1640     return i;
1641 }
1642
1643 bool XMLElement::BoolAttribute(const char* name, bool defaultValue) const
1644 {
1645     bool b = defaultValue;
1646     QueryBoolAttribute(name, &b);
1647     return b;
1648 }
1649
1650 double XMLElement::DoubleAttribute(const char* name, double defaultValue) const
1651 {
```



```
1652     double d = defaultValue;
1653     QueryDoubleAttribute(name, &d);
1654     return d;
1655 }
1656
1657 float XElement::FloatAttribute(const char* name, float defaultValue) const
1658 {
1659     float f = defaultValue;
1660     QueryFloatAttribute(name, &f);
1661     return f;
1662 }
1663
1664 const char* XElement::GetText() const
1665 {
1666     /* skip comment node */
1667     const XMLNode* node = FirstChild();
1668     while (node) {
1669         if (node->ToComment()) {
1670             node = node->NextSibling();
1671             continue;
1672         }
1673         break;
1674     }
1675
1676     if ( node && node->ToText() ) {
1677         return node->Value();
1678     }
1679     return 0;
1680 }
1681
1682
1683 void XElement::SetText( const char* inText )
1684 {
1685     if ( FirstChild() && FirstChild()->ToText() )
1686         FirstChild()->SetValue( inText );
1687     else {
1688         XMLText*      theText = GetDocument()->NewText( inText );
1689         InsertFirstChild( theText );
1690     }
1691 }
1692
1693
1694 void XElement::SetText( int v )
1695 {
1696     char buf[BUF_SIZE];
1697     XMLUtil::ToStr( v, buf, BUF_SIZE );
1698     SetText( buf );
1699 }
1700
```

```
1701
1702 void XMLElement::SetText( unsigned v )
1703 {
1704     char buf[BUF_SIZE];
1705     XMLUtil::ToStr( v, buf, BUF_SIZE );
1706     SetText( buf );
1707 }
1708
1709
1710 void XMLElement::SetText(int64_t v)
1711 {
1712     char buf[BUF_SIZE];
1713     XMLUtil::ToStr(v, buf, BUF_SIZE);
1714     SetText(buf);
1715 }
1716
1717 void XMLElement::SetText(uint64_t v) {
1718     char buf[BUF_SIZE];
1719     XMLUtil::ToStr(v, buf, BUF_SIZE);
1720     SetText(buf);
1721 }
1722
1723
1724 void XMLElement::SetText( bool v )
1725 {
1726     char buf[BUF_SIZE];
1727     XMLUtil::ToStr( v, buf, BUF_SIZE );
1728     SetText( buf );
1729 }
1730
1731
1732 void XMLElement::SetText( float v )
1733 {
1734     char buf[BUF_SIZE];
1735     XMLUtil::ToStr( v, buf, BUF_SIZE );
1736     SetText( buf );
1737 }
1738
1739
1740 void XMLElement::SetText( double v )
1741 {
1742     char buf[BUF_SIZE];
1743     XMLUtil::ToStr( v, buf, BUF_SIZE );
1744     SetText( buf );
1745 }
1746
1747
1748 XMLError XMLElement::QueryIntText( int* ival ) const
1749 {
```

```
1750     if ( FirstChild() && FirstChild()->ToText() ) {
1751         const char* t = FirstChild()->Value();
1752         if ( XMLUtil::ToInt( t, ival ) ) {
1753             return XML_SUCCESS;
1754         }
1755         return XML_CAN_NOT_CONVERT_TEXT;
1756     }
1757     return XML_NO_TEXT_NODE;
1758 }
1759
1760
1761 XMLError XMLElement::QueryUnsignedText( unsigned* uval ) const
1762 {
1763     if ( FirstChild() && FirstChild()->ToText() ) {
1764         const char* t = FirstChild()->Value();
1765         if ( XMLUtil::ToUnsigned( t, uval ) ) {
1766             return XML_SUCCESS;
1767         }
1768         return XML_CAN_NOT_CONVERT_TEXT;
1769     }
1770     return XML_NO_TEXT_NODE;
1771 }
1772
1773
1774 XMLError XMLElement::QueryInt64Text(int64_t* ival) const
1775 {
1776     if (FirstChild() && FirstChild()->ToText()) {
1777         const char* t = FirstChild()->Value();
1778         if (XMLUtil::ToInt64(t, ival)) {
1779             return XML_SUCCESS;
1780         }
1781         return XML_CAN_NOT_CONVERT_TEXT;
1782     }
1783     return XML_NO_TEXT_NODE;
1784 }
1785
1786
1787 XMLError XMLElement::QueryUnsigned64Text(uint64_t* uval) const
1788 {
1789     if(FirstChild() && FirstChild()->ToText()) {
1790         const char* t = FirstChild()->Value();
1791         if(XMLUtil::ToUnsigned64(t, uval)) {
1792             return XML_SUCCESS;
1793         }
1794         return XML_CAN_NOT_CONVERT_TEXT;
1795     }
1796     return XML_NO_TEXT_NODE;
1797 }
1798
```

```
1799
1800 XMLError XMLElement::QueryBoolText( bool* bval ) const
1801 {
1802     if ( FirstChild() && FirstChild()->ToText() ) {
1803         const char* t = FirstChild()->Value();
1804         if ( XMLUtil::ToBool( t, bval ) ) {
1805             return XML_SUCCESS;
1806         }
1807         return XML_CAN_NOT_CONVERT_TEXT;
1808     }
1809     return XML_NO_TEXT_NODE;
1810 }
1811
1812
1813 XMLError XMLElement::QueryDoubleText( double* dval ) const
1814 {
1815     if ( FirstChild() && FirstChild()->ToText() ) {
1816         const char* t = FirstChild()->Value();
1817         if ( XMLUtil::ToDouble( t, dval ) ) {
1818             return XML_SUCCESS;
1819         }
1820         return XML_CAN_NOT_CONVERT_TEXT;
1821     }
1822     return XML_NO_TEXT_NODE;
1823 }
1824
1825
1826 XMLError XMLElement::QueryFloatText( float* fval ) const
1827 {
1828     if ( FirstChild() && FirstChild()->ToText() ) {
1829         const char* t = FirstChild()->Value();
1830         if ( XMLUtil::ToFloat( t, fval ) ) {
1831             return XML_SUCCESS;
1832         }
1833         return XML_CAN_NOT_CONVERT_TEXT;
1834     }
1835     return XML_NO_TEXT_NODE;
1836 }
1837
1838 int XMLElement::IntText(int defaultValue) const
1839 {
1840     int i = defaultValue;
1841     QueryIntText(&i);
1842     return i;
1843 }
1844
1845 unsigned XMLElement::UnsignedText(unsigned defaultValue) const
1846 {
1847     unsigned i = defaultValue;
```

```
1848     QueryUnsignedText(&i);
1849     return i;
1850 }
1851
1852 int64_t XElement::Int64Text(int64_t defaultValue) const
1853 {
1854     int64_t i = defaultValue;
1855     QueryInt64Text(&i);
1856     return i;
1857 }
1858
1859 uint64_t XElement::Unsigned64Text(uint64_t defaultValue) const
1860 {
1861     uint64_t i = defaultValue;
1862     QueryUnsigned64Text(&i);
1863     return i;
1864 }
1865
1866 bool XElement::BoolText(bool defaultValue) const
1867 {
1868     bool b = defaultValue;
1869     QueryBoolText(&b);
1870     return b;
1871 }
1872
1873 double XElement::DoubleText(double defaultValue) const
1874 {
1875     double d = defaultValue;
1876     QueryDoubleText(&d);
1877     return d;
1878 }
1879
1880 float XElement::FloatText(float defaultValue) const
1881 {
1882     float f = defaultValue;
1883     QueryFloatText(&f);
1884     return f;
1885 }
1886
1887
1888 XMLAttribute* XElement::FindOrCreateAttribute( const char* name )
1889 {
1890     XMLAttribute* last = 0;
1891     XMLAttribute* attrib = 0;
1892     for( attrib = _rootAttribute;
1893         attrib;
1894         last = attrib, attrib = attrib->_next ) {
1895         if ( XMLUtil::StringEqual( attrib->Name(), name ) ) {
1896             break;
```

```
1897     }
1898 }
1899 if ( !attrib ) {
1900     attrib = CreateAttribute();
1901     TIXMLASSERT( attrib );
1902     if ( last ) {
1903         TIXMLASSERT( last->_next == 0 );
1904         last->_next = attrib;
1905     }
1906     else {
1907         TIXMLASSERT( _rootAttribute == 0 );
1908         _rootAttribute = attrib;
1909     }
1910     attrib->SetName( name );
1911 }
1912 return attrib;
1913 }
1914
1915
1916 void XMLElement::DeleteAttribute( const char* name )
1917 {
1918     XMLAttribute* prev = 0;
1919     for( XMLAttribute* a=_rootAttribute; a; a=a->_next ) {
1920         if ( XMLUtil::StringEqual( name, a->Name() ) ) {
1921             if ( prev ) {
1922                 prev->_next = a->_next;
1923             }
1924             else {
1925                 _rootAttribute = a->_next;
1926             }
1927             DeleteAttribute( a );
1928             break;
1929         }
1930         prev = a;
1931     }
1932 }
1933
1934
1935 char* XMLElement::ParseAttributes( char* p, int* curLineNumPtr )
1936 {
1937     XMLAttribute* prevAttribute = 0;
1938
1939     // Read the attributes.
1940     while( p ) {
1941         p = XMLUtil::SkipWhiteSpace( p, curLineNumPtr );
1942         if ( !(*p) ) {
1943             _document->SetError( XML_ERROR_PARSING_ELEMENT, _parseLineNum, "XMLElement
name=%s", Name() );
1944             return 0;

```

```

1945     }
1946
1947     // attribute.
1948     if (XMLUtil::IsNameStartChar( (unsigned char) *p )) {
1949         XMLAttribute* attrib = CreateAttribute();
1950         TIXMLASSERT( attrib );
1951         attrib->_parseLineNum = _document->_parseCurLineNum;
1952
1953         const int attrLineNum = attrib->_parseLineNum;
1954
1955         p = attrib->ParseDeep( p, _document->ProcessEntities(), curLineNumPtr );
1956         if ( !p || Attribute( attrib->Name() ) ) {
1957             DeleteAttribute( attrib );
1958             _document->SetError( XML_ERROR_PARSING_ATTRIBUTE, attrLineNum, "XMLElement
name=%s", Name() );
1959             return 0;
1960         }
1961         // There is a minor bug here: if the attribute in the source xml
1962         // document is duplicated, it will not be detected and the
1963         // attribute will be doubly added. However, tracking the 'prevAttribute'
1964         // avoids re-scanning the attribute list. Preferring performance for
1965         // now, may reconsider in the future.
1966         if ( prevAttribute ) {
1967             TIXMLASSERT( prevAttribute->_next == 0 );
1968             prevAttribute->_next = attrib;
1969         }
1970         else {
1971             TIXMLASSERT( _rootAttribute == 0 );
1972             _rootAttribute = attrib;
1973         }
1974         prevAttribute = attrib;
1975     }
1976     // end of the tag
1977     else if ( *p == '>' ) {
1978         ++p;
1979         break;
1980     }
1981     // end of the tag
1982     else if ( *p == '/' && *(p+1) == '>' ) {
1983         _closingType = CLOSED;
1984         return p+2; // done; sealed element.
1985     }
1986     else {
1987         _document->SetError( XML_ERROR_PARSING_ELEMENT, _parseLineNum, 0 );
1988         return 0;
1989     }
1990 }
1991 return p;
1992 }

```

```
1993
1994 void XElement::DeleteAttribute( XMLAttribute* attribute )
1995 {
1996     if ( attribute == 0 ) {
1997         return;
1998     }
1999     MemPool* pool = attribute->_memPool;
2000     attribute->~XMLAttribute();
2001     pool->Free( attribute );
2002 }
2003
2004 XMLAttribute* XElement::CreateAttribute()
2005 {
2006     TIXMLASSERT( sizeof( XMLAttribute ) == _document->_attributePool.ItemSize() );
2007     XMLAttribute* attrib = new ( _document->_attributePool.Alloc() ) XMLAttribute();
2008     TIXMLASSERT( attrib );
2009     attrib->_memPool = &_document->_attributePool;
2010     attrib->_memPool->SetTracked();
2011     return attrib;
2012 }
2013
2014
2015 XElement* XElement::InsertNewChildElement(const char* name)
2016 {
2017     XElement* node = _document->NewElement(name);
2018     return InsertEndChild(node) ? node : 0;
2019 }
2020
2021 XMLComment* XElement::InsertNewComment(const char* comment)
2022 {
2023     XMLComment* node = _document->NewComment(comment);
2024     return InsertEndChild(node) ? node : 0;
2025 }
2026
2027 XMLText* XElement::InsertNewText(const char* text)
2028 {
2029     XMLText* node = _document->NewText(text);
2030     return InsertEndChild(node) ? node : 0;
2031 }
2032
2033 XMLDeclaration* XElement::InsertNewDeclaration(const char* text)
2034 {
2035     XMLDeclaration* node = _document->NewDeclaration(text);
2036     return InsertEndChild(node) ? node : 0;
2037 }
2038
2039 XMLUnknown* XElement::InsertNewUnknown(const char* text)
2040 {
2041     XMLUnknown* node = _document->NewUnknown(text);
```



```

2042     return InsertEndChild(node) ? node : 0;
2043 }
2044
2045
2046
2047 //
2048 // <ele></ele>
2049 // <ele>foo<b>bar</b></ele>
2050 //
2051 char* XMLElement::ParseDeep( char* p, StrPair* parentEndTag, int* curLineNumPtr )
2052 {
2053     // Read the element name.
2054     p = XMLUtil::SkipWhiteSpace( p, curLineNumPtr );
2055
2056     // The closing element is the </element> form. It is
2057     // parsed just like a regular element then deleted from
2058     // the DOM.
2059     if ( *p == '/' ) {
2060         _closingType = CLOSING;
2061         ++p;
2062     }
2063
2064     p = _value.ParseName( p );
2065     if ( _value.Empty() ) {
2066         return 0;
2067     }
2068
2069     p = ParseAttributes( p, curLineNumPtr );
2070     if ( !p || !*p || _closingType != OPEN ) {
2071         return p;
2072     }
2073
2074     p = XMLNode::ParseDeep( p, parentEndTag, curLineNumPtr );
2075     return p;
2076 }
2077
2078
2079
2080 XMLNode* XMLElement::ShallowClone( XMLDocument* doc ) const
2081 {
2082     if ( !doc ) {
2083         doc = _document;
2084     }
2085     XMLElement* element = doc->NewElement( Value() );           // fixme: this will always
allocate memory. Intern?
2086     for( const XMLAttribute* a=FirstAttribute(); a;a->Next() ) {
2087         element->SetAttribute( a->Name(), a->Value() );           // fixme: this will always allocate
memory. Intern?
2088     }

```

```

2089     return element;
2090 }
2091
2092
2093 bool XMLElement::ShallowEqual( const XMLNode* compare ) const
2094 {
2095     TIXMLASSERT( compare );
2096     const XMLElement* other = compare->ToElement();
2097     if ( other && XMLUtil::StringEqual( other->Name(), Name() ) ) {
2098
2099         const XMLAttribute* a=FirstAttribute();
2100         const XMLAttribute* b=other->FirstAttribute();
2101
2102         while ( a && b ) {
2103             if ( !XMLUtil::StringEqual( a->Value(), b->Value() ) ) {
2104                 return false;
2105             }
2106             a = a->Next();
2107             b = b->Next();
2108         }
2109         if ( a || b ) {
2110             // different count
2111             return false;
2112         }
2113         return true;
2114     }
2115     return false;
2116 }
2117
2118
2119 bool XMLElement::Accept( XMLVisitor* visitor ) const
2120 {
2121     TIXMLASSERT( visitor );
2122     if ( visitor->VisitEnter( *this, _rootAttribute ) ) {
2123         for ( const XMLNode* node=FirstChild(); node; node=node->NextSibling() ) {
2124             if ( !node->Accept( visitor ) ) {
2125                 break;
2126             }
2127         }
2128     }
2129     return visitor->VisitExit( *this );
2130 }
2131
2132
2133 // ----- XMLDocument ----- //
2134
2135 // Warning: List must match 'enum XMLError'
2136 const char* XMLDocument::_errorNames[XML_ERROR_COUNT] = {
2137     "XML_SUCCESS",

```

```

2138 "XML_NO_ATTRIBUTE",
2139 "XML_WRONG_ATTRIBUTE_TYPE",
2140 "XML_ERROR_FILE_NOT_FOUND",
2141 "XML_ERROR_FILE_COULD_NOT_BE_OPENED",
2142 "XML_ERROR_FILE_READ_ERROR",
2143 "XML_ERROR_PARSING_ELEMENT",
2144 "XML_ERROR_PARSING_ATTRIBUTE",
2145 "XML_ERROR_PARSING_TEXT",
2146 "XML_ERROR_PARSING_CDATA",
2147 "XML_ERROR_PARSING_COMMENT",
2148 "XML_ERROR_PARSING_DECLARATION",
2149 "XML_ERROR_PARSING_UNKNOWN",
2150 "XML_ERROR_EMPTY_DOCUMENT",
2151 "XML_ERROR_MISMATCHED_ELEMENT",
2152 "XML_ERROR_PARSING",
2153 "XML_CAN_NOT_CONVERT_TEXT",
2154 "XML_NO_TEXT_NODE",
2155     "XML_ELEMENT_DEPTH_EXCEEDED"
2156 };
2157
2158
2159 XmlDocument::XmlDocument( bool processEntities, Whitespace whitespaceMode ) :
2160     XmlNode( 0 ),
2161     _writeBOM( false ),
2162     _processEntities( processEntities ),
2163     _errorID(XML_SUCCESS),
2164     _whitespaceMode( whitespaceMode ),
2165     _errorStr(),
2166     _errorLineNum( 0 ),
2167     _charBuffer( 0 ),
2168     _parseCurLineNum( 0 ),
2169     _parsingDepth(0),
2170     _unlinked(),
2171     _elementPool(),
2172     _attributePool(),
2173     _textPool(),
2174     _commentPool()
2175 {
2176     // avoid VC++ C4355 warning about 'this' in initializer list (C4355 is off by default in VS2012+)
2177     _document = this;
2178 }
2179
2180
2181 XmlDocument::~~XmlDocument()
2182 {
2183     Clear();
2184 }
2185
2186

```

```

2187 void XMLDocument::MarkInUse(const XMLNode* const node)
2188 {
2189     TIXMLASSERT(node);
2190     TIXMLASSERT(node->_parent == 0);
2191
2192     for (int i = 0; i < _unlinked.Size(); ++i) {
2193         if (node == _unlinked[i]) {
2194             _unlinked.SwapRemove(i);
2195             break;
2196         }
2197     }
2198 }
2199
2200 void XMLDocument::Clear()
2201 {
2202     DeleteChildren();
2203     while( _unlinked.Size()) {
2204         DeleteNode(_unlinked[0]); // Will remove from _unlinked as part of delete.
2205     }
2206
2207 #ifdef TINYXML2_DEBUG
2208     const bool hadError = Error();
2209 #endif
2210     ClearError();
2211
2212     delete [] _charBuffer;
2213     _charBuffer = 0;
2214     _parsingDepth = 0;
2215
2216 #if 0
2217     _textPool.Trace( "text" );
2218     _elementPool.Trace( "element" );
2219     _commentPool.Trace( "comment" );
2220     _attributePool.Trace( "attribute" );
2221 #endif
2222
2223 #ifdef TINYXML2_DEBUG
2224     if ( !hadError ) {
2225         TIXMLASSERT( _elementPool.CurrentAllocs() == _elementPool.Untracked() );
2226         TIXMLASSERT( _attributePool.CurrentAllocs() == _attributePool.Untracked() );
2227         TIXMLASSERT( _textPool.CurrentAllocs() == _textPool.Untracked() );
2228         TIXMLASSERT( _commentPool.CurrentAllocs() == _commentPool.Untracked() );
2229     }
2230 #endif
2231 }
2232
2233
2234 void XMLDocument::DeepCopy(XMLDocument* target) const
2235 {

```

```
2236     TIXMLASSERT(target);
2237     if (target == this) {
2238         return; // technically success - a no-op.
2239     }
2240
2241     target->Clear();
2242     for (const XMLNode* node = this->FirstChild(); node; node = node->NextSibling()) {
2243         target->InsertEndChild(node->DeepClone(target));
2244     }
2245 }
2246
2247 XMLElement* XMLDocument::NewElement( const char* name )
2248 {
2249     XMLElement* ele = CreateUnlinkedNode<XMLElement>(_elementPool );
2250     ele->SetName( name );
2251     return ele;
2252 }
2253
2254
2255 XMLComment* XMLDocument::NewComment( const char* str )
2256 {
2257     XMLComment* comment = CreateUnlinkedNode<XMLComment>(_commentPool );
2258     comment->SetValue( str );
2259     return comment;
2260 }
2261
2262
2263 XMLText* XMLDocument::NewText( const char* str )
2264 {
2265     XMLText* text = CreateUnlinkedNode<XMLText>(_textPool );
2266     text->SetValue( str );
2267     return text;
2268 }
2269
2270
2271 XMLDeclaration* XMLDocument::NewDeclaration( const char* str )
2272 {
2273     XMLDeclaration* dec = CreateUnlinkedNode<XMLDeclaration>(_commentPool );
2274     dec->SetValue( str ? str : "xml version=\"1.0\" encoding=\"UTF-8\"");
2275     return dec;
2276 }
2277
2278
2279 XMLUnknown* XMLDocument::NewUnknown( const char* str )
2280 {
2281     XMLUnknown* unk = CreateUnlinkedNode<XMLUnknown>(_commentPool );
2282     unk->SetValue( str );
2283     return unk;
2284 }
```

```

2285
2286 static FILE* callfopen( const char* filepath, const char* mode )
2287 {
2288     TIXMLASSERT( filepath );
2289     TIXMLASSERT( mode );
2290     #if defined(_MSC_VER) && (_MSC_VER >= 1400) && (!defined WINCE)
2291         FILE* fp = 0;
2292         const errno_t err = fopen_s( &fp, filepath, mode );
2293         if ( err ) {
2294             return 0;
2295         }
2296     #else
2297         FILE* fp = fopen( filepath, mode );
2298     #endif
2299     return fp;
2300 }
2301
2302 void XMLDocument::DeleteNode( XMLNode* node ) {
2303     TIXMLASSERT( node );
2304     TIXMLASSERT( node->_document == this );
2305     if ( node->_parent ) {
2306         node->_parent->DeleteChild( node );
2307     }
2308     else {
2309         // Isn't in the tree.
2310         // Use the parent delete.
2311         // Also, we need to mark it tracked: we 'know'
2312         // it was never used.
2313         node->_memPool->SetTracked();
2314         // Call the static XMLNode version:
2315         XMLNode::DeleteNode( node );
2316     }
2317 }
2318
2319
2320 XMLError XMLDocument::LoadFile( const char* filename )
2321 {
2322     if ( !filename ) {
2323         TIXMLASSERT( false );
2324         SetError( XML_ERROR_FILE_COULD_NOT_BE_OPENED, 0, "filename=<null>" );
2325         return _errorID;
2326     }
2327
2328     Clear();
2329     FILE* fp = callfopen( filename, "rb" );
2330     if ( !fp ) {
2331         SetError( XML_ERROR_FILE_NOT_FOUND, 0, "filename=%s", filename );
2332         return _errorID;
2333     }

```

```

2334 LoadFile( fp );
2335 fclose( fp );
2336 return _errorID;
2337 }
2338
2339 XMLError XMLDocument::LoadFile( FILE* fp )
2340 {
2341     Clear();
2342
2343     TIXML_FSEEK( fp, 0, SEEK_SET );
2344     if ( fgetc( fp ) == EOF && ferror( fp ) != 0 ) {
2345         SetError( XML_ERROR_FILE_READ_ERROR, 0, 0 );
2346         return _errorID;
2347     }
2348
2349     TIXML_FSEEK( fp, 0, SEEK_END );
2350
2351     unsigned long long filelength;
2352     {
2353         const long long fileLengthSigned = TIXML_FTELL( fp );
2354         TIXML_FSEEK( fp, 0, SEEK_SET );
2355         if ( fileLengthSigned == -1L ) {
2356             SetError( XML_ERROR_FILE_READ_ERROR, 0, 0 );
2357             return _errorID;
2358         }
2359         TIXMLASSERT( fileLengthSigned >= 0 );
2360         filelength = static_cast<unsigned long long>(fileLengthSigned);
2361     }
2362
2363     const size_t maxSizeT = static_cast<size_t>(-1);
2364     // We'll do the comparison as an unsigned long long, because that's guaranteed to be at
2365     // least 8 bytes, even on a 32-bit platform.
2366     if ( filelength >= static_cast<unsigned long long>(maxSizeT) ) {
2367         // Cannot handle files which won't fit in buffer together with null terminator
2368         SetError( XML_ERROR_FILE_READ_ERROR, 0, 0 );
2369         return _errorID;
2370     }
2371
2372     if ( filelength == 0 ) {
2373         SetError( XML_ERROR_EMPTY_DOCUMENT, 0, 0 );
2374         return _errorID;
2375     }
2376
2377     const size_t size = static_cast<size_t>(filelength);
2378     TIXMLASSERT( _charBuffer == 0 );
2379     _charBuffer = new char[size+1];
2380     const size_t read = fread( _charBuffer, 1, size, fp );
2381     if ( read != size ) {
2382         SetError( XML_ERROR_FILE_READ_ERROR, 0, 0 );

```

```

2383     return _errorID;
2384 }
2385
2386 _charBuffer[size] = 0;
2387
2388 Parse();
2389 return _errorID;
2390 }
2391
2392
2393 XMLError XMLDocument::SaveFile( const char* filename, bool compact )
2394 {
2395     if ( !filename ) {
2396         TIXMLASSERT( false );
2397         SetError( XML_ERROR_FILE_COULD_NOT_BE_OPENED, 0, "filename=<null>" );
2398         return _errorID;
2399     }
2400
2401     FILE* fp = callfopen( filename, "w" );
2402     if ( !fp ) {
2403         SetError( XML_ERROR_FILE_COULD_NOT_BE_OPENED, 0, "filename=%s", filename );
2404         return _errorID;
2405     }
2406     SaveFile(fp, compact);
2407     fclose( fp );
2408     return _errorID;
2409 }
2410
2411
2412 XMLError XMLDocument::SaveFile( FILE* fp, bool compact )
2413 {
2414     // Clear any error from the last save, otherwise it will get reported
2415     // for *this* call.
2416     ClearError();
2417     XMLPrinter stream( fp, compact );
2418     Print( &stream );
2419     return _errorID;
2420 }
2421
2422
2423 XMLError XMLDocument::Parse( const char* xml, size_t nBytes )
2424 {
2425     Clear();
2426
2427     if ( nBytes == 0 || !xml || !*xml ) {
2428         SetError( XML_ERROR_EMPTY_DOCUMENT, 0, 0 );
2429         return _errorID;
2430     }
2431     if ( nBytes == static_cast<size_t>(-1) ) {

```



```
2432     nBytes = strlen( xml );
2433 }
2434 TIXMLASSERT( _charBuffer == 0 );
2435 _charBuffer = new char[ nBytes+1 ];
2436 memcpy( _charBuffer, xml, nBytes );
2437 _charBuffer[nBytes] = 0;
2438
2439 Parse();
2440 if ( Error() ) {
2441     // clean up now essentially dangling memory.
2442     // and the parse fail can put objects in the
2443     // pools that are dead and inaccessible.
2444     DeleteChildren();
2445     _elementPool.Clear();
2446     _attributePool.Clear();
2447     _textPool.Clear();
2448     _commentPool.Clear();
2449 }
2450 return _errorID;
2451 }
2452
2453
2454 void XMLDocument::Print( XMLPrinter* streamer ) const
2455 {
2456     if ( streamer ) {
2457         Accept( streamer );
2458     }
2459     else {
2460         XMLPrinter stdoutStreamer( stdout );
2461         Accept( &stdoutStreamer );
2462     }
2463 }
2464
2465
2466 void XMLDocument::ClearError() {
2467     _errorID = XML_SUCCESS;
2468     _errorLineNum = 0;
2469     _errorStr.Reset();
2470 }
2471
2472
2473 void XMLDocument::SetError( XMLError error, int lineNumber, const char* format, ... )
2474 {
2475     TIXMLASSERT( error >= 0 && error < XML_ERROR_COUNT );
2476     _errorID = error;
2477     _errorLineNum = lineNumber;
2478     _errorStr.Reset();
2479
2480     const size_t BUFFER_SIZE = 1000;
```

```

2481     char* buffer = new char[BUFFER_SIZE];
2482
2483     TIXMLASSERT(sizeof(error) <= sizeof(int));
2484     TIXML_SNPRINTF(buffer, BUFFER_SIZE, "Error=%s ErrorID=%d (0x%x) Line number=%d",
ErrorIDToName(error), int(error), int(error), lineNum);
2485
2486     if (format) {
2487         size_t len = strlen(buffer);
2488         TIXML_SNPRINTF(buffer + len, BUFFER_SIZE - len, ": ");
2489         len = strlen(buffer);
2490
2491         va_list va;
2492         va_start(va, format);
2493         TIXML_VSNPRINTF(buffer + len, BUFFER_SIZE - len, format, va);
2494         va_end(va);
2495     }
2496     _errorStr.SetStr(buffer);
2497     delete[] buffer;
2498 }
2499
2500
2501 /*static*/ const char* XMLDocument::ErrorIDToName(XMLError errorID)
2502 {
2503     TIXMLASSERT( errorID >= 0 && errorID < XML_ERROR_COUNT );
2504     const char* errorName = _errorNames[errorID];
2505     TIXMLASSERT( errorName && errorName[0] );
2506     return errorName;
2507 }
2508
2509 const char* XMLDocument::ErrorStr() const
2510 {
2511     return _errorStr.Empty() ? "" : _errorStr.GetStr();
2512 }
2513
2514
2515 void XMLDocument::PrintError() const
2516 {
2517     printf("%s\n", ErrorStr());
2518 }
2519
2520 const char* XMLDocument::ErrorName() const
2521 {
2522     return ErrorIDToName(_errorID);
2523 }
2524
2525 void XMLDocument::Parse()
2526 {
2527     TIXMLASSERT( NoChildren() ); // Clear() must have been called previously
2528     TIXMLASSERT( !_charBuffer );

```

```

2529     _parseCurLineNum = 1;
2530     _parseLineNum = 1;
2531     char* p = _charBuffer;
2532     p = XMLUtil::SkipWhiteSpace( p, &_parseCurLineNum );
2533     p = const_cast<char*>( XMLUtil::ReadBOM( p, &_writeBOM ) );
2534     if ( !*p ) {
2535         SetError( XML_ERROR_EMPTY_DOCUMENT, 0, 0 );
2536         return;
2537     }
2538     ParseDeep(p, 0, &_parseCurLineNum );
2539 }
2540
2541 void XMLDocument::PushDepth()
2542 {
2543     _parsingDepth++;
2544     if ( _parsingDepth == TINYXML2_MAX_ELEMENT_DEPTH ) {
2545         SetError(XML_ELEMENT_DEPTH_EXCEEDED, _parseCurLineNum, "Element nesting is too deep."
2546     );
2547     }
2548 }
2549 void XMLDocument::PopDepth()
2550 {
2551     TIXMLASSERT(_parsingDepth > 0);
2552     --_parsingDepth;
2553 }
2554
2555 XMLPrinter::XMLPrinter( FILE* file, bool compact, int depth ) :
2556     _elementJustOpened( false ),
2557     _stack(),
2558     _firstElement( true ),
2559     _fp( file ),
2560     _depth( depth ),
2561     _textDepth( -1 ),
2562     _processEntities( true ),
2563     _compactMode( compact ),
2564     _buffer()
2565 {
2566     for( int i=0; i<ENTITY_RANGE; ++i ) {
2567         _entityFlag[i] = false;
2568         _restrictedEntityFlag[i] = false;
2569     }
2570     for( int i=0; i<NUM_ENTITIES; ++i ) {
2571         const char entityValue = entities[i].value;
2572         const unsigned char flagIndex = static_cast<unsigned char>(entityValue);
2573         TIXMLASSERT( flagIndex < ENTITY_RANGE );
2574         _entityFlag[flagIndex] = true;
2575     }
2576     _restrictedEntityFlag[static_cast<unsigned char>('&')] = true;

```

```

2577 _restrictedEntityFlag[static_cast<unsigned char>('<')] = true;
2578 _restrictedEntityFlag[static_cast<unsigned char>('>')] = true;    // not required, but consistency is
    nice
2579 _buffer.Push( 0 );
2580 }
2581
2582
2583 void XMLPrinter::Print( const char* format, ... )
2584 {
2585     va_list va;
2586     va_start( va, format );
2587
2588     if ( _fp ) {
2589         vfprintf( _fp, format, va );
2590     }
2591     else {
2592         const int len = TIXML_VSCPRINTF( format, va );
2593         // Close out and re-start the va-args
2594         va_end( va );
2595         TIXMLASSERT( len >= 0 );
2596         va_start( va, format );
2597         TIXMLASSERT( _buffer.Size() > 0 && _buffer[_buffer.Size() - 1] == 0 );
2598         char* p = _buffer.PushArr( len ) - 1;    // back up over the null terminator.
2599         TIXML_VSNPRINTF( p, len+1, format, va );
2600     }
2601     va_end( va );
2602 }
2603
2604
2605 void XMLPrinter::Write( const char* data, size_t size )
2606 {
2607     if ( _fp ) {
2608         fwrite ( data , sizeof(char), size, _fp);
2609     }
2610     else {
2611         char* p = _buffer.PushArr( static_cast<int>(size) ) - 1;    // back up over the null terminator.
2612         memcpy( p, data, size );
2613         p[size] = 0;
2614     }
2615 }
2616
2617
2618 void XMLPrinter::Putc( char ch )
2619 {
2620     if ( _fp ) {
2621         fputc ( ch, _fp);
2622     }
2623     else {
2624         char* p = _buffer.PushArr( sizeof(char) ) - 1;    // back up over the null terminator.

```

```

2625     p[0] = ch;
2626     p[1] = 0;
2627 }
2628 }
2629
2630
2631 void XMLPrinter::PrintSpace( int depth )
2632 {
2633     for( int i=0; i<depth; ++i ) {
2634         Write( "  " );
2635     }
2636 }
2637
2638
2639 void XMLPrinter::PrintString( const char* p, bool restricted )
2640 {
2641     // Look for runs of bytes between entities to print.
2642     const char* q = p;
2643
2644     if ( _processEntities ) {
2645         const bool* flag = restricted ? _restrictedEntityFlag : _entityFlag;
2646         while ( *q ) {
2647             TIXMLASSERT( p <= q );
2648             // Remember, char is sometimes signed. (How many times has that bitten me?)
2649             if ( *q > 0 && *q < ENTITY_RANGE ) {
2650                 // Check for entities. If one is found, flush
2651                 // the stream up until the entity, write the
2652                 // entity, and keep looking.
2653                 if ( flag[static_cast<unsigned char>(*q)] ) {
2654                     while ( p < q ) {
2655                         const size_t delta = q - p;
2656                         const int toPrint = ( INT_MAX < delta ) ? INT_MAX : static_cast<int>(delta);
2657                         Write( p, toPrint );
2658                         p += toPrint;
2659                     }
2660                     bool entityPatternPrinted = false;
2661                     for( int i=0; i<NUM_ENTITIES; ++i ) {
2662                         if ( entities[i].value == *q ) {
2663                             Putc( '&' );
2664                             Write( entities[i].pattern, entities[i].length );
2665                             Putc( ';' );
2666                             entityPatternPrinted = true;
2667                             break;
2668                         }
2669                     }
2670                     if ( !entityPatternPrinted ) {
2671                         // TIXMLASSERT( entityPatternPrinted ) causes gcc -Wunused-but-set-variable in release
2672                         TIXMLASSERT( false );
2673                     }

```

```

2674         ++p;
2675     }
2676 }
2677 ++q;
2678 TIXMLASSERT( p <= q );
2679 }
2680 // Flush the remaining string. This will be the entire
2681 // string if an entity wasn't found.
2682 if ( p < q ) {
2683     const size_t delta = q - p;
2684     const int toPrint = ( INT_MAX < delta ) ? INT_MAX : static_cast<int>(delta);
2685     Write( p, toPrint );
2686 }
2687 }
2688 else {
2689     Write( p );
2690 }
2691 }
2692
2693
2694 void XMLPrinter::PushHeader( bool writeBOM, bool writeDec )
2695 {
2696     if ( writeBOM ) {
2697         static const unsigned char bom[] = { TIXML_UTF_LEAD_0, TIXML_UTF_LEAD_1, TIXML_UTF_LEAD_2, 0
2698     };
2699     Write( reinterpret_cast< const char* >( bom ) );
2700 }
2701 if ( writeDec ) {
2702     PushDeclaration( "xml version=\"1.0\"" );
2703 }
2704 }
2705
2706 void XMLPrinter::PrepareForNewNode( bool compactMode )
2707 {
2708     SealElementIfJustOpened();
2709
2710     if ( compactMode ) {
2711         return;
2712     }
2713
2714     if ( _firstElement ) {
2715         PrintSpace( _depth );
2716     } else if ( _textDepth < 0 ) {
2717         Putc( '\n' );
2718         PrintSpace( _depth );
2719     }
2720
2721     _firstElement = false;
2722 }

```

```
2722
2723 void XMLPrinter::OpenElement( const char* name, bool compactMode )
2724 {
2725     PrepareForNewNode( compactMode );
2726     _stack.Push( name );
2727
2728     Write ( "<" );
2729     Write ( name );
2730
2731     _elementJustOpened = true;
2732     ++_depth;
2733 }
2734
2735
2736 void XMLPrinter::PushAttribute( const char* name, const char* value )
2737 {
2738     TIXMLASSERT( _elementJustOpened );
2739     Putc ( ' ' );
2740     Write( name );
2741     Write( "=\"" );
2742     PrintString( value, false );
2743     Putc ( "\"" );
2744 }
2745
2746
2747 void XMLPrinter::PushAttribute( const char* name, int v )
2748 {
2749     char buf[BUF_SIZE];
2750     XMLUtil::ToStr( v, buf, BUF_SIZE );
2751     PushAttribute( name, buf );
2752 }
2753
2754
2755 void XMLPrinter::PushAttribute( const char* name, unsigned v )
2756 {
2757     char buf[BUF_SIZE];
2758     XMLUtil::ToStr( v, buf, BUF_SIZE );
2759     PushAttribute( name, buf );
2760 }
2761
2762
2763 void XMLPrinter::PushAttribute(const char* name, int64_t v)
2764 {
2765     char buf[BUF_SIZE];
2766     XMLUtil::ToStr(v, buf, BUF_SIZE);
2767     PushAttribute(name, buf);
2768 }
2769
2770
```

```
2771 void XMLPrinter::PushAttribute(const char* name, uint64_t v)
2772 {
2773     char buf[BUF_SIZE];
2774     XMLUtil::ToStr(v, buf, BUF_SIZE);
2775     PushAttribute(name, buf);
2776 }
2777
2778
2779 void XMLPrinter::PushAttribute( const char* name, bool v )
2780 {
2781     char buf[BUF_SIZE];
2782     XMLUtil::ToStr( v, buf, BUF_SIZE );
2783     PushAttribute( name, buf );
2784 }
2785
2786
2787 void XMLPrinter::PushAttribute( const char* name, double v )
2788 {
2789     char buf[BUF_SIZE];
2790     XMLUtil::ToStr( v, buf, BUF_SIZE );
2791     PushAttribute( name, buf );
2792 }
2793
2794
2795 void XMLPrinter::CloseElement( bool compactMode )
2796 {
2797     --_depth;
2798     const char* name = _stack.Pop();
2799
2800     if ( _elementJustOpened ) {
2801         Write( ">" );
2802     }
2803     else {
2804         if ( _textDepth < 0 && !compactMode ) {
2805             Putc( '\n' );
2806             PrintSpace( _depth );
2807         }
2808         Write ( "</" );
2809         Write ( name );
2810         Write ( ">" );
2811     }
2812
2813     if ( _textDepth == _depth ) {
2814         _textDepth = -1;
2815     }
2816     if ( _depth == 0 && !compactMode ) {
2817         Putc( '\n' );
2818     }
2819     _elementJustOpened = false;
```



```
2820 }
2821
2822
2823 void XMLPrinter::SealElementIfJustOpened()
2824 {
2825     if ( !_elementJustOpened ) {
2826         return;
2827     }
2828     _elementJustOpened = false;
2829     Putc( '>' );
2830 }
2831
2832
2833 void XMLPrinter::PushText( const char* text, bool cdata )
2834 {
2835     _textDepth = _depth-1;
2836
2837     SealElementIfJustOpened();
2838     if ( cdata ) {
2839         Write( "<![CDATA[" );
2840         Write( text );
2841         Write( "]">" );
2842     }
2843     else {
2844         PrintString( text, true );
2845     }
2846 }
2847
2848
2849 void XMLPrinter::PushText( int64_t value )
2850 {
2851     char buf[BUF_SIZE];
2852     XMLUtil::ToStr( value, buf, BUF_SIZE );
2853     PushText( buf, false );
2854 }
2855
2856
2857 void XMLPrinter::PushText( uint64_t value )
2858 {
2859     char buf[BUF_SIZE];
2860     XMLUtil::ToStr( value, buf, BUF_SIZE );
2861     PushText( buf, false );
2862 }
2863
2864
2865 void XMLPrinter::PushText( int value )
2866 {
2867     char buf[BUF_SIZE];
2868     XMLUtil::ToStr( value, buf, BUF_SIZE );
```

```
2869     PushText( buf, false );
2870 }
2871
2872
2873 void XMLPrinter::PushText( unsigned value )
2874 {
2875     char buf[BUF_SIZE];
2876     XMLUtil::ToStr( value, buf, BUF_SIZE );
2877     PushText( buf, false );
2878 }
2879
2880
2881 void XMLPrinter::PushText( bool value )
2882 {
2883     char buf[BUF_SIZE];
2884     XMLUtil::ToStr( value, buf, BUF_SIZE );
2885     PushText( buf, false );
2886 }
2887
2888
2889 void XMLPrinter::PushText( float value )
2890 {
2891     char buf[BUF_SIZE];
2892     XMLUtil::ToStr( value, buf, BUF_SIZE );
2893     PushText( buf, false );
2894 }
2895
2896
2897 void XMLPrinter::PushText( double value )
2898 {
2899     char buf[BUF_SIZE];
2900     XMLUtil::ToStr( value, buf, BUF_SIZE );
2901     PushText( buf, false );
2902 }
2903
2904
2905 void XMLPrinter::PushComment( const char* comment )
2906 {
2907     PrepareForNewNode( _compactMode );
2908
2909     Write( "<!--" );
2910     Write( comment );
2911     Write( "-->" );
2912 }
2913
2914
2915 void XMLPrinter::PushDeclaration( const char* value )
2916 {
2917     PrepareForNewNode( _compactMode );
```

```
2918
2919     Write( "<?" );
2920     Write( value );
2921     Write( ">" );
2922 }
2923
2924
2925 void XMLPrinter::PushUnknown( const char* value )
2926 {
2927     PrepareForNewNode( _compactMode );
2928
2929     Write( "<!" );
2930     Write( value );
2931     Putc( '>' );
2932 }
2933
2934
2935 bool XMLPrinter::VisitEnter( const XMLDocument& doc )
2936 {
2937     _processEntities = doc.ProcessEntities();
2938     if ( doc.HasBOM() ) {
2939         PushHeader( true, false );
2940     }
2941     return true;
2942 }
2943
2944
2945 bool XMLPrinter::VisitEnter( const XMLElement& element, const XMLAttribute* attribute )
2946 {
2947     const XMLElement* parentElem = 0;
2948     if ( element.Parent() ) {
2949         parentElem = element.Parent()->ToElement();
2950     }
2951     const bool compactMode = parentElem ? CompactMode( *parentElem ) : _compactMode;
2952     OpenElement( element.Name(), compactMode );
2953     while ( attribute ) {
2954         PushAttribute( attribute->Name(), attribute->Value() );
2955         attribute = attribute->Next();
2956     }
2957     return true;
2958 }
2959
2960
2961 bool XMLPrinter::VisitExit( const XMLElement& element )
2962 {
2963     CloseElement( CompactMode(element) );
2964     return true;
2965 }
2966
```

```
2967
2968 bool XMLPrinter::Visit( const XMLText& text )
2969 {
2970     PushText( text.Value(), text.CData() );
2971     return true;
2972 }
2973
2974
2975 bool XMLPrinter::Visit( const XMLComment& comment )
2976 {
2977     PushComment( comment.Value() );
2978     return true;
2979 }
2980
2981 bool XMLPrinter::Visit( const XMLDeclaration& declaration )
2982 {
2983     PushDeclaration( declaration.Value() );
2984     return true;
2985 }
2986
2987
2988 bool XMLPrinter::Visit( const XMLUnknown& unknown )
2989 {
2990     PushUnknown( unknown.Value() );
2991     return true;
2992 }
2993
2994 } // namespace tinyxml2
2995
```

```
1  /*
2  Original code by Lee Thomason (www.grinninglizard.com)
3
4  This software is provided 'as-is', without any express or implied
5  warranty. In no event will the authors be held liable for any
6  damages arising from the use of this software.
7
8  Permission is granted to anyone to use this software for any
9  purpose, including commercial applications, and to alter it and
10 redistribute it freely, subject to the following restrictions:
11
12 1. The origin of this software must not be misrepresented; you must
13 not claim that you wrote the original software. If you use this
14 software in a product, an acknowledgment in the product documentation
15 would be appreciated but is not required.
16
17 2. Altered source versions must be plainly marked as such, and
18 must not be misrepresented as being the original software.
19
20 3. This notice may not be removed or altered from any source
21 distribution.
22 */
23
24 #ifndef TINYXML2_INCLUDED
25 #define TINYXML2_INCLUDED
26
27 #if defined(ANDROID_NDK) || defined(__BORLANDC__) || defined(__QNXNTO__)
28 # include <ctype.h>
29 # include <limits.h>
30 # include <stdio.h>
31 # include <stdlib.h>
32 # include <string.h>
33 # if defined(__PS3__)
34 # include <stddef.h>
35 # endif
36 #else
37 # include <cctype>
38 # include <climits>
39 # include <cstdio>
40 # include <cstdlib>
41 # include <cstring>
42 #endif
43 #include <stdint.h>
44
45 /*
46 TODO: intern strings instead of allocation.
```

```

47 */
48 /*
49     gcc:
50     g++ -Wall -DTINYXML2_DEBUG tinyxml2.cpp xmltest.cpp -o gccxmltest.exe
51
52     Formatting, Artistic Style:
53     AStyle.exe --style=1tbs --indent-switches --break-closing-brackets --indent-preprocessor
tinyxml2.cpp tinyxml2.h
54 */
55
56 #if defined( _DEBUG ) || defined ( __DEBUG__ )
57 #   ifndef TINYXML2_DEBUG
58 #       define TINYXML2_DEBUG
59 #   endif
60 #endif
61
62 #ifdef _MSC_VER
63 #   pragma warning(push)
64 #   pragma warning(disable: 4251)
65 #endif
66
67 #ifdef _WIN32
68 #   ifdef TINYXML2_EXPORT
69 #       define TINYXML2_LIB __declspec(dllexport)
70 #   elif defined(TINYXML2_IMPORT)
71 #       define TINYXML2_LIB __declspec(dllimport)
72 #   else
73 #       define TINYXML2_LIB
74 #   endif
75 #elif __GNUC__ >= 4
76 #   define TINYXML2_LIB __attribute__((visibility("default")))
77 #else
78 #   define TINYXML2_LIB
79 #endif
80
81
82 #if !defined(TIXMLASSERT)
83 #if defined(TINYXML2_DEBUG)
84 #   if defined(_MSC_VER)
85 #       // "(void)0," is for suppressing C4127 warning in "assert(false)", "assert(true)" and the like
86 #       define TIXMLASSERT( x )    do { if ( !(void)0,(x)) { __debugbreak(); } } while(false)
87 #   elif defined(ANDROID_NDK)
88 #       include <android/log.h>
89 #       define TIXMLASSERT( x )    do { if ( !(x)) { __android_log_assert( "assert", "grinliz", "ASSERT in '%s' at %d.", __FILE__, __LINE__ ); } } while(false)
90 #   else
91 #       include <assert.h>
92 #       define TIXMLASSERT         assert
93 #   endif

```

```

94  #else
95  #  define TIXMLASSERT( x )          do {} while(false)
96  #endif
97  #endif
98
99  /* Versioning, past 1.0.14:
100     http://semver.org/
101  */
102  static const int TIXML2_MAJOR_VERSION = 9;
103  static const int TIXML2_MINOR_VERSION = 0;
104  static const int TIXML2_PATCH_VERSION = 0;
105
106  #define TINYXML2_MAJOR_VERSION 9
107  #define TINYXML2_MINOR_VERSION 0
108  #define TINYXML2_PATCH_VERSION 0
109
110  // A fixed element depth limit is problematic. There needs to be a
111  // limit to avoid a stack overflow. However, that limit varies per
112  // system, and the capacity of the stack. On the other hand, it's a trivial
113  // attack that can result from ill, malicious, or even correctly formed XML,
114  // so there needs to be a limit in place.
115  static const int TINYXML2_MAX_ELEMENT_DEPTH = 500;
116
117  namespace tinyxml2
118  {
119  class XMLDocument;
120  class XMLElement;
121  class XMLAttribute;
122  class XMLComment;
123  class XMLText;
124  class XMLDeclaration;
125  class XMLUnknown;
126  class XMLPrinter;
127
128  /*
129     A class that wraps strings. Normally stores the start and end
130     pointers into the XML file itself, and will apply normalization
131     and entity translation if actually read. Can also store (and memory
132     manage) a traditional char[]
133
134     Isn't clear why TINYXML2_LIB is needed; but seems to fix #719
135  */
136  class TINYXML2_LIB StrPair
137  {
138  public:
139     enum Mode {
140         NEEDS_ENTITY_PROCESSING      = 0x01,
141         NEEDS_NEWLINE_NORMALIZATION  = 0x02,
142         NEEDS_WHITESPACE_COLLAPSING  = 0x04,

```

```

143
144     TEXT_ELEMENT                = NEEDS_ENTITY_PROCESSING |
NEEDS_NEWLINE_NORMALIZATION,
145     TEXT_ELEMENT_LEAVE_ENTITIES    = NEEDS_NEWLINE_NORMALIZATION,
146     ATTRIBUTE_NAME                = 0,
147     ATTRIBUTE_VALUE                = NEEDS_ENTITY_PROCESSING |
NEEDS_NEWLINE_NORMALIZATION,
148     ATTRIBUTE_VALUE_LEAVE_ENTITIES = NEEDS_NEWLINE_NORMALIZATION,
149     COMMENT                        = NEEDS_NEWLINE_NORMALIZATION
150 };
151
152 StrPair() : _flags( 0 ), _start( 0 ), _end( 0 ) {}
153 ~StrPair();
154
155 void Set( char* start, char* end, int flags ) {
156     TIXMLASSERT( start );
157     TIXMLASSERT( end );
158     Reset();
159     _start = start;
160     _end   = end;
161     _flags = flags | NEEDS_FLUSH;
162 }
163
164 const char* GetStr();
165
166 bool Empty() const {
167     return _start == _end;
168 }
169
170 void SetInternedStr( const char* str ) {
171     Reset();
172     _start = const_cast<char*>(str);
173 }
174
175 void SetStr( const char* str, int flags=0 );
176
177 char* ParseText( char* in, const char* endTag, int strFlags, int* curLineNumPtr );
178 char* ParseName( char* in );
179
180 void TransferTo( StrPair* other );
181 void Reset();
182
183 private:
184 void CollapseWhitespace();
185
186 enum {
187     NEEDS_FLUSH = 0x100,
188     NEEDS_DELETE = 0x200
189 };

```



```

190
191     int _flags;
192     char* _start;
193     char* _end;
194
195     StrPair( const StrPair& other ); // not supported
196     void operator=( const StrPair& other ); // not supported, use TransferTo()
197 };
198
199
200 /*
201     A dynamic array of Plain Old Data. Doesn't support constructors, etc.
202     Has a small initial memory pool, so that low or no usage will not
203     cause a call to new/delete
204 */
205 template <class T, int INITIAL_SIZE>
206 class DynArray
207 {
208 public:
209     DynArray() :
210         _mem( _pool ),
211         _allocated( INITIAL_SIZE ),
212         _size( 0 )
213     {
214     }
215
216     ~DynArray() {
217         if ( _mem != _pool ) {
218             delete [] _mem;
219         }
220     }
221
222     void Clear() {
223         _size = 0;
224     }
225
226     void Push( T t ) {
227         TIXMLASSERT( _size < INT_MAX );
228         EnsureCapacity( _size+1 );
229         _mem[_size] = t;
230         ++_size;
231     }
232
233     T* PushArr( int count ) {
234         TIXMLASSERT( count >= 0 );
235         TIXMLASSERT( _size <= INT_MAX - count );
236         EnsureCapacity( _size+count );
237         T* ret = &_mem[_size];
238         _size += count;

```

```

239     return ret;
240 }
241
242 T Pop() {
243     TIXMLASSERT( _size > 0 );
244     --_size;
245     return _mem[_size];
246 }
247
248 void PopArr( int count ) {
249     TIXMLASSERT( _size >= count );
250     _size -= count;
251 }
252
253 bool Empty() const {
254     return _size == 0;
255 }
256
257 T& operator[](int i)
{
258     TIXMLASSERT( i >= 0 && i < _size );
259     return _mem[i];
260 }
261
262 const T& operator[](int i) const {
263     TIXMLASSERT( i >= 0 && i < _size );
264     return _mem[i];
265 }
266
267 const T& PeekTop() const {
268     TIXMLASSERT( _size > 0 );
269     return _mem[ _size - 1];
270 }
271
272 int Size() const {
273     TIXMLASSERT( _size >= 0 );
274     return _size;
275 }
276
277 int Capacity() const {
278     TIXMLASSERT( _allocated >= INITIAL_SIZE );
279     return _allocated;
280 }
281
282 void SwapRemove(int i) {
283     TIXMLASSERT( i >= 0 && i < _size);
284     TIXMLASSERT( _size > 0);
285     _mem[i] = _mem[_size - 1];
286     --_size;

```

```

287     }
288
289     const T* Mem() const          {
290         TIXMLASSERT( _mem );
291         return _mem;
292     }
293
294     T* Mem() {
295         TIXMLASSERT( _mem );
296         return _mem;
297     }
298
299 private:
300     DynArray( const DynArray& ); // not supported
301     void operator=( const DynArray& ); // not supported
302
303     void EnsureCapacity( int cap ) {
304         TIXMLASSERT( cap > 0 );
305         if ( cap > _allocated ) {
306             TIXMLASSERT( cap <= INT_MAX / 2 );
307             const int newAllocated = cap * 2;
308             T* newMem = new T[newAllocated];
309             TIXMLASSERT( newAllocated >= _size );
310             memcpy( newMem, _mem, sizeof(T)*_size ); // warning: not using constructors, only works for
PODs
311             if ( _mem != _pool ) {
312                 delete [] _mem;
313             }
314             _mem = newMem;
315             _allocated = newAllocated;
316         }
317     }
318
319     T* _mem;
320     T _pool[INITIAL_SIZE];
321     int _allocated;          // objects allocated
322     int _size;               // number objects in use
323 };
324
325
326 /*
327     Parent virtual class of a pool for fast allocation
328     and deallocation of objects.
329 */
330 class MemPool
331 {
332 public:
333     MemPool() {}
334     virtual ~MemPool() {}

```

```

335
336     virtual int ItemSize() const = 0;
337     virtual void* Alloc() = 0;
338     virtual void Free( void* ) = 0;
339     virtual void SetTracked() = 0;
340 };
341
342
343 /*
344     Template child class to create pools of the correct type.
345 */
346 template< int ITEM_SIZE >
347 class MemPoolT : public MemPool
348 {
349 public:
350     MemPoolT() : _blockPtrs(), _root(0), _currentAllocs(0), _nAllocs(0), _maxAllocs(0), _nUntracked(0)
351     {}
352     ~MemPoolT() {
353         MemPoolT< ITEM_SIZE >::Clear();
354     }
355
356     void Clear() {
357         // Delete the blocks.
358         while( !_blockPtrs.Empty()) {
359             Block* lastBlock = _blockPtrs.Pop();
360             delete lastBlock;
361         }
362         _root = 0;
363         _currentAllocs = 0;
364         _nAllocs = 0;
365         _maxAllocs = 0;
366         _nUntracked = 0;
367     }
368
369     virtual int ItemSize() const {
370         return ITEM_SIZE;
371     }
372
373     int CurrentAllocs() const {
374         return _currentAllocs;
375     }
376
377     virtual void* Alloc() {
378         if ( !_root ) {
379             // Need a new block.
380             Block* block = new Block;
381             _blockPtrs.Push( block );
382
383             Item* blockItems = block->items;
384             for( int i = 0; i < ITEMS_PER_BLOCK - 1; ++i ) {

```

```

383     blockItems[i].next = &(blockItems[i + 1]);
384 }
385     blockItems[ITEMS_PER_BLOCK - 1].next = 0;
386     _root = blockItems;
387 }
388     Item* const result = _root;
389     TIXMLASSERT( result != 0 );
390     _root = _root->next;
391
392     ++_currentAllocs;
393     if ( _currentAllocs > _maxAllocs ) {
394         _maxAllocs = _currentAllocs;
395     }
396     ++_nAllocs;
397     ++_nUntracked;
398     return result;
399 }
400
401 virtual void Free( void* mem ) {
402     if ( !mem ) {
403         return;
404     }
405     --_currentAllocs;
406     Item* item = static_cast<Item*>( mem );
407 #ifdef TINYXML2_DEBUG
408     memset( item, 0xfe, sizeof( *item ) );
409 #endif
410     item->next = _root;
411     _root = item;
412 }
413 void Trace( const char* name ) {
414     printf( "Mempool %s watermark=%d [%dk] current=%d size=%d nAlloc=%d blocks=%d\n",
415         name, _maxAllocs, _maxAllocs * ITEM_SIZE / 1024, _currentAllocs,
416         ITEM_SIZE, _nAllocs, _blockPtrs.Size() );
417 }
418
419 void SetTracked() {
420     --_nUntracked;
421 }
422
423 int Untracked() const {
424     return _nUntracked;
425 }
426
427 // This number is perf sensitive. 4k seems like a good tradeoff on my machine.
428 // The test file is large, 170k.
429 // Release:      VS2010 gcc(no opt)
430 //      1k:      4000
431 //      2k:      4000

```

```

432 //      4k:      3900      21000
433 //      16k: 5200
434 //      32k: 4300
435 //      64k: 4000      21000
436 // Declared public because some compilers do not accept to use ITEMS_PER_BLOCK
437 // in private part if ITEMS_PER_BLOCK is private
438 enum { ITEMS_PER_BLOCK = (4 * 1024) / ITEM_SIZE };
439
440 private:
441     MemPoolT( const MemPoolT& ); // not supported
442     void operator=( const MemPoolT& ); // not supported
443
444     union Item {
445         Item* next;
446         char  itemData[ITEM_SIZE];
447     };
448     struct Block {
449         Item items[ITEMS_PER_BLOCK];
450     };
451     DynArray< Block*, 10 > _blockPtrs;
452     Item* _root;
453
454     int _currentAllocs;
455     int _nAllocs;
456     int _maxAllocs;
457     int _nUntracked;
458 };
459
460
461
462 /**
463     Implements the interface to the "Visitor pattern" (see the Accept() method.)
464     If you call the Accept() method, it requires being passed a XMLVisitor
465     class to handle callbacks. For nodes that contain other nodes (Document, Element)
466     you will get called with a VisitEnter/VisitExit pair. Nodes that are always leafs
467     are simply called with Visit().
468
469     If you return 'true' from a Visit method, recursive parsing will continue. If you return
470     false, <b>no children of this node or its siblings</b> will be visited.
471
472     All flavors of Visit methods have a default implementation that returns 'true' (continue
473     visiting). You need to only override methods that are interesting to you.
474
475     Generally Accept() is called on the XMLDocument, although all nodes support visiting.
476
477     You should never change the document from a callback.
478
479     @sa XMLNode::Accept()
480 */

```

```

481 class TINYXML2_LIB XMLVisitor
482 {
483 public:
484     virtual ~XMLVisitor() {}
485
486     /// Visit a document.
487     virtual bool VisitEnter( const XMLDocument& /*doc*/ )      {
488         return true;
489     }
490     /// Visit a document.
491     virtual bool VisitExit( const XMLDocument& /*doc*/ )      {
492         return true;
493     }
494
495     /// Visit an element.
496     virtual bool VisitEnter( const XMLElement& /*element*/, const XMLAttribute* /*firstAttribute*/ ) {
497         return true;
498     }
499     /// Visit an element.
500     virtual bool VisitExit( const XMLElement& /*element*/ )   {
501         return true;
502     }
503
504     /// Visit a declaration.
505     virtual bool Visit( const XMLDeclaration& /*declaration*/ ) {
506         return true;
507     }
508     /// Visit a text node.
509     virtual bool Visit( const XMLText& /*text*/ )              {
510         return true;
511     }
512     /// Visit a comment node.
513     virtual bool Visit( const XMLComment& /*comment*/ )        {
514         return true;
515     }
516     /// Visit an unknown node.
517     virtual bool Visit( const XMLUnknown& /*unknown*/ )        {
518         return true;
519     }
520 };
521
522 // WARNING: must match XMLDocument::_errorNames[]
523 enum XMLError {
524     XML_SUCCESS = 0,
525     XML_NO_ATTRIBUTE,
526     XML_WRONG_ATTRIBUTE_TYPE,
527     XML_ERROR_FILE_NOT_FOUND,
528     XML_ERROR_FILE_COULD_NOT_BE_OPENED,
529     XML_ERROR_FILE_READ_ERROR,

```

```

530 XML_ERROR_PARSING_ELEMENT,
531 XML_ERROR_PARSING_ATTRIBUTE,
532 XML_ERROR_PARSING_TEXT,
533 XML_ERROR_PARSING_CDATA,
534 XML_ERROR_PARSING_COMMENT,
535 XML_ERROR_PARSING_DECLARATION,
536 XML_ERROR_PARSING_UNKNOWN,
537 XML_ERROR_EMPTY_DOCUMENT,
538 XML_ERROR_MISMATCHED_ELEMENT,
539 XML_ERROR_PARSING,
540 XML_CAN_NOT_CONVERT_TEXT,
541 XML_NO_TEXT_NODE,
542     XML_ELEMENT_DEPTH_EXCEEDED,
543
544     XML_ERROR_COUNT
545 };
546
547
548 /*
549     Utility functionality.
550 */
551 class TINYXML2_LIB XMLUtil
552 {
553 public:
554     static const char* SkipWhiteSpace( const char* p, int* curLineNumPtr )    {
555         TIXMLASSERT( p );
556
557         while( IsWhiteSpace(*p) ) {
558             if (curLineNumPtr && *p == '\n') {
559                 ++(*curLineNumPtr);
560             }
561             ++p;
562         }
563         TIXMLASSERT( p );
564         return p;
565     }
566     static char* SkipWhiteSpace( char* const p, int* curLineNumPtr ) {
567         return const_cast<char*>( SkipWhiteSpace( const_cast<const char*>(p), curLineNumPtr ) );
568     }
569
570     // Anything in the high order range of UTF-8 is assumed to not be whitespace. This isn't
571     // correct, but simple, and usually works.
572     static bool IsWhiteSpace( char p ) {
573         return !IsUTF8Continuation(p) && isspace( static_cast<unsigned char>(p) );
574     }
575
576     inline static bool IsNameStartChar( unsigned char ch ) {
577         if ( ch >= 128 ) {
578             // This is a heuristic guess in attempt to not implement Unicode-aware isalpha()

```



```

579         return true;
580     }
581     if ( isalpha( ch ) ) {
582         return true;
583     }
584     return ch == ':' || ch == '_';
585 }
586
587 inline static bool IsNameChar( unsigned char ch ) {
588     return IsNameStartChar( ch )
589         || isdigit( ch )
590         || ch == '.'
591         || ch == '-';
592 }
593
594 inline static bool IsPrefixHex( const char* p ) {
595     p = SkipWhiteSpace(p, 0);
596     return p && *p == '0' && ( *(p + 1) == 'x' || *(p + 1) == 'X');
597 }
598
599 inline static bool StringEqual( const char* p, const char* q, int nChar=INT_MAX ) {
600     if ( p == q ) {
601         return true;
602     }
603     TIXMLASSERT( p );
604     TIXMLASSERT( q );
605     TIXMLASSERT( nChar >= 0 );
606     return strncmp( p, q, nChar ) == 0;
607 }
608
609 inline static bool IsUTF8Continuation( const char p ) {
610     return ( p & 0x80 ) != 0;
611 }
612
613 static const char* ReadBOM( const char* p, bool* hasBOM );
614 // p is the starting location,
615 // the UTF-8 value of the entity will be placed in value, and length filled in.
616 static const char* GetCharacterRef( const char* p, char* value, int* length );
617 static void ConvertUTF32ToUTF8( unsigned long input, char* output, int* length );
618
619 // converts primitive types to strings
620 static void ToStr( int v, char* buffer, int bufferSize );
621 static void ToStr( unsigned v, char* buffer, int bufferSize );
622 static void ToStr( bool v, char* buffer, int bufferSize );
623 static void ToStr( float v, char* buffer, int bufferSize );
624 static void ToStr( double v, char* buffer, int bufferSize );
625     static void ToStr(int64_t v, char* buffer, int bufferSize);
626 static void ToStr(uint64_t v, char* buffer, int bufferSize);
627

```

```

628 // converts strings to primitive types
629 static bool ToInt( const char* str, int* value );
630 static bool ToUnsigned( const char* str, unsigned* value );
631 static bool ToBool( const char* str, bool* value );
632 static bool ToFloat( const char* str, float* value );
633 static bool ToDouble( const char* str, double* value );
634     static bool ToInt64(const char* str, int64_t* value);
635 static bool ToUnsigned64(const char* str, uint64_t* value);
636 // Changes what is serialized for a boolean value.
637 // Default to "true" and "false". Shouldn't be changed
638 // unless you have a special testing or compatibility need.
639 // Be careful: static, global, & not thread safe.
640 // Be sure to set static const memory as parameters.
641 static void SetBoolSerialization(const char* writeTrue, const char* writeFalse);
642
643 private:
644     static const char* writeBoolTrue;
645     static const char* writeBoolFalse;
646 };
647
648
649 /** XMLNode is a base class for every object that is in the
650     XML Document Object Model (DOM), except XMLAttributes.
651     Nodes have siblings, a parent, and children which can
652     be navigated. A node is always in a XMLDocument.
653     The type of a XMLNode can be queried, and it can
654     be cast to its more defined type.
655
656     A XMLDocument allocates memory for all its Nodes.
657     When the XMLDocument gets deleted, all its Nodes
658     will also be deleted.
659
660     @verbatim
661     A Document can contain:    Element (container or leaf)
662                               Comment (leaf)
663                               Unknown (leaf)
664                               Declaration( leaf )
665
666     An Element can contain:    Element (container or leaf)
667                               Text (leaf)
668                               Attributes (not on tree)
669                               Comment (leaf)
670                               Unknown (leaf)
671
672     @endverbatim
673 */
674 class TINYXML2_LIB XMLNode
675 {
676     friend class XMLDocument;

```

```

677     friend class XMLElement;
678 public:
679
680     /// Get the XMLDocument that owns this XMLNode.
681     const XMLDocument* GetDocument() const {
682         TIXMLASSERT( _document );
683         return _document;
684     }
685     /// Get the XMLDocument that owns this XMLNode.
686     XMLDocument* GetDocument() {
687         TIXMLASSERT( _document );
688         return _document;
689     }
690
691     /// Safely cast to an Element, or null.
692     virtual XMLElement* ToElement() {
693         return 0;
694     }
695     /// Safely cast to Text, or null.
696     virtual XMLText* ToText() {
697         return 0;
698     }
699     /// Safely cast to a Comment, or null.
700     virtual XMLComment* ToComment() {
701         return 0;
702     }
703     /// Safely cast to a Document, or null.
704     virtual XMLDocument* ToDocument() {
705         return 0;
706     }
707     /// Safely cast to a Declaration, or null.
708     virtual XMLDeclaration* ToDeclaration() {
709         return 0;
710     }
711     /// Safely cast to an Unknown, or null.
712     virtual XMLUnknown* ToUnknown() {
713         return 0;
714     }
715
716     virtual const XMLElement* ToElement() const {
717         return 0;
718     }
719     virtual const XMLText* ToText() const {
720         return 0;
721     }
722     virtual const XMLComment* ToComment() const {
723         return 0;
724     }
725     virtual const XMLDocument* ToDocument() const {

```

```

726     return 0;
727 }
728 virtual const XMLDeclaration* ToDeclaration() const {
729     return 0;
730 }
731 virtual const XMLUnknown* ToUnknown() const {
732     return 0;
733 }
734
735 /** The meaning of 'value' changes for the specific type.
736     @verbatim
737     Document:    empty (NULL is returned, not an empty string)
738     Element:    name of the element
739     Comment:    the comment text
740     Unknown:    the tag contents
741     Text:       the text string
742     @endverbatim
743 */
744 const char* Value() const;
745
746 /** Set the Value of an XML node.
747     @sa Value()
748 */
749 void SetValue( const char* val, bool staticMem=false );
750
751 /// Gets the line number the node is in, if the document was parsed from a file.
752 int GetLineNum() const { return _parseLineNum; }
753
754 /// Get the parent of this node on the DOM.
755 const XMLNode* Parent() const {
756     return _parent;
757 }
758
759 XMLNode* Parent() {
760     return _parent;
761 }
762
763 /// Returns true if this node has no children.
764 bool NoChildren() const {
765     return !_firstChild;
766 }
767
768 /// Get the first child node, or null if none exists.
769 const XMLNode* FirstChild() const {
770     return _firstChild;
771 }
772
773 XMLNode* FirstChild() {
774     return _firstChild;

```

```

775 }
776
777 /** Get the first child element, or optionally the first child
778     element with the specified name.
779 */
780 const XElement* FirstChildElement( const char* name = 0 ) const;
781
782 XElement* FirstChildElement( const char* name = 0 ) {
783     return const_cast<XElement*>(const_cast<const XMLNode*>(this)->FirstChildElement( name ));
784 }
785
786 /// Get the last child node, or null if none exists.
787 const XMLNode* LastChild() const {
788     return _lastChild;
789 }
790
791 XMLNode* LastChild() {
792     return _lastChild;
793 }
794
795 /** Get the last child element or optionally the last child
796     element with the specified name.
797 */
798 const XElement* LastChildElement( const char* name = 0 ) const;
799
800 XElement* LastChildElement( const char* name = 0 ) {
801     return const_cast<XElement*>(const_cast<const XMLNode*>(this)->LastChildElement(name) );
802 }
803
804 /// Get the previous (left) sibling node of this node.
805 const XMLNode* PreviousSibling() const {
806     return _prev;
807 }
808
809 XMLNode* PreviousSibling() {
810     return _prev;
811 }
812
813 /// Get the previous (left) sibling element of this node, with an optionally supplied name.
814 const XElement* PreviousSiblingElement( const char* name = 0 ) const;
815
816 XElement* PreviousSiblingElement( const char* name = 0 ) {
817     return const_cast<XElement*>(const_cast<const XMLNode*>(this)->PreviousSiblingElement(
name ));
818 }
819
820 /// Get the next (right) sibling node of this node.
821 const XMLNode* NextSibling() const {
822     return _next;

```

```

823     }
824
825     XMLNode* NextSibling()                {
826         return _next;
827     }
828
829     /// Get the next (right) sibling element of this node, with an optionally supplied name.
830     const XMLElement* NextSiblingElement( const char* name = 0 ) const;
831
832     XMLElement* NextSiblingElement( const char* name = 0 )    {
833         return const_cast<XMLElement*>(const_cast<const XMLNode*>(this)->NextSiblingElement( name )
834     );
835     }
836
837     /**
838         Add a child node as the last (right) child.
839         If the child node is already part of the document,
840         it is moved from its old location to the new location.
841         Returns the addThis argument or 0 if the node does not
842         belong to the same document.
843     */
844     XMLNode* InsertEndChild( XMLNode* addThis );
845
846     XMLNode* LinkEndChild( XMLNode* addThis )    {
847         return InsertEndChild( addThis );
848     }
849
850     /**
851         Add a child node as the first (left) child.
852         If the child node is already part of the document,
853         it is moved from its old location to the new location.
854         Returns the addThis argument or 0 if the node does not
855         belong to the same document.
856     */
857     XMLNode* InsertFirstChild( XMLNode* addThis );
858
859     /**
860         Add a node after the specified child node.
861         If the child node is already part of the document,
862         it is moved from its old location to the new location.
863         Returns the addThis argument or 0 if the afterThis node
864         is not a child of this node, or if the node does not
865         belong to the same document.
866     */
867     XMLNode* InsertAfterChild( XMLNode* afterThis, XMLNode* addThis );
868
869     /**
870         Delete all the children of this node.
871     */
872     void DeleteChildren();

```

```

871  /**
872      Delete a child of this node.
873  */
874  void DeleteChild( XMLNode* node );
875
876  /**
877      Make a copy of this node, but not its children.
878      You may pass in a Document pointer that will be
879      the owner of the new Node. If the 'document' is
880      null, then the node returned will be allocated
881      from the current Document. (this->GetDocument())
882
883      Note: if called on a XMLDocument, this will return null.
884  */
885  virtual XMLNode* ShallowClone( XMLDocument* document ) const = 0;
886
887  /**
888      Make a copy of this node and all its children.
889
890      If the 'target' is null, then the nodes will
891      be allocated in the current document. If 'target'
892      is specified, the memory will be allocated in the
893      specified XMLDocument.
894
895      NOTE: This is probably not the correct tool to
896      copy a document, since XMLDocuments can have multiple
897      top level XMLNodes. You probably want to use
898      XMLDocument::DeepCopy()
899  */
900  XMLNode* DeepClone( XMLDocument* target ) const;
901
902  /**
903      Test if 2 nodes are the same, but don't test children.
904      The 2 nodes do not need to be in the same Document.
905
906      Note: if called on a XMLDocument, this will return false.
907  */
908  virtual bool ShallowEqual( const XMLNode* compare ) const = 0;
909
910  /** Accept a hierarchical visit of the nodes in the TinyXML-2 DOM. Every node in the
911      XML tree will be conditionally visited and the host will be called back
912      via the XMLVisitor interface.
913
914      This is essentially a SAX interface for TinyXML-2. (Note however it doesn't re-parse
915      the XML for the callbacks, so the performance of TinyXML-2 is unchanged by using this
916      interface versus any other.)
917
918      The interface has been based on ideas from:
919

```

- <http://www.saxproject.org/>
- <http://c2.com/cgi/wiki?HierarchicalVisitorPattern>

Which are both good references for "visiting".

An example of using Accept():

@verbatim

XMLPrinter printer;

tinyxmlDoc.Accept(&printer);

const char* xmlcstr = printer.CStr();

@endverbatim

*/

virtual bool Accept(XMLVisitor* visitor) const = 0;

/**

Set user data into the XMLNode. TinyXML-2 in
no way processes or interprets user data.
It is initially 0.

*/

void SetUserData(void* userData) { _userData = userData; }

/**

Get user data set into the XMLNode. TinyXML-2 in
no way processes or interprets user data.
It is initially 0.

*/

void* GetUserData() const { return _userData; }

protected:

explicit XMLNode(XMLDocument*);

virtual ~XMLNode();

virtual char* ParseDeep(char* p, StrPair* parentEndTag, int* curLineNumPtr);

XMLDocument* _document;

XMLNode* _parent;

mutable StrPair _value;

int _parseLineNum;

XMLNode* _firstChild;

XMLNode* _lastChild;

XMLNode* _prev;

XMLNode* _next;

void*

_userData;

private:


```

968     MemPool*      _memPool;
969     void Unlink( XMLNode* child );
970     static void DeleteNode( XMLNode* node );
971     void InsertChildPreamble( XMLNode* insertThis ) const;
972     const XMLElement* ToElementWithName( const char* name ) const;
973
974     XMLNode( const XMLNode& ); // not supported
975     XMLNode& operator=( const XMLNode& ); // not supported
976 };
977
978
979 /** XML text.
980
981     Note that a text node can have child element nodes, for example:
982     @verbatim
983     <root>This is <b>bold</b></root>
984     @endverbatim
985
986     A text node can have 2 ways to output the next. "normal" output
987     and CDATA. It will default to the mode it was parsed from the XML file and
988     you generally want to leave it alone, but you can change the output mode with
989     SetCDATA() and query it with CData().
990 */
991 class TINYXML2_LIB XMLText : public XMLNode
992 {
993     friend class XMLDocument;
994 public:
995     virtual bool Accept( XMLVisitor* visitor ) const;
996
997     virtual XMLText* ToText()          {
998         return this;
999     }
1000     virtual const XMLText* ToText() const {
1001         return this;
1002     }
1003
1004     /// Declare whether this should be CDATA or standard text.
1005     void SetCDATA( bool isCDATA )      {
1006         _isCDATA = isCDATA;
1007     }
1008     /// Returns true if this is a CDATA text element.
1009     bool CData() const                  {
1010         return _isCDATA;
1011     }
1012
1013     virtual XMLNode* ShallowClone( XMLDocument* document ) const;
1014     virtual bool ShallowEqual( const XMLNode* compare ) const;
1015
1016 protected:

```

```

1017     explicit XMLText( XMLDocument* doc ) : XMLNode( doc ), _isCData( false )    {}
1018     virtual ~XMLText()
1019 {}
1020     char* ParseDeep( char* p, StrPair* parentEndTag, int* curLineNumPtr );
1021
1022 private:
1023     bool _isCData;
1024
1025     XMLText( const XMLText& );    // not supported
1026     XMLText& operator=( const XMLText& );    // not supported
1027 };
1028
1029
1030 /** An XML Comment. */
1031 class TINYXML2_LIB XMLComment : public XMLNode
1032 {
1033     friend class XMLDocument;
1034 public:
1035     virtual XMLComment*    ToComment()                {
1036         return this;
1037     }
1038     virtual const XMLComment* ToComment() const        {
1039         return this;
1040     }
1041
1042     virtual bool Accept( XMLVisitor* visitor ) const;
1043
1044     virtual XMLNode* ShallowClone( XMLDocument* document ) const;
1045     virtual bool ShallowEqual( const XMLNode* compare ) const;
1046
1047 protected:
1048     explicit XMLComment( XMLDocument* doc );
1049     virtual ~XMLComment();
1050
1051     char* ParseDeep( char* p, StrPair* parentEndTag, int* curLineNumPtr);
1052
1053 private:
1054     XMLComment( const XMLComment& );    // not supported
1055     XMLComment& operator=( const XMLComment& );    // not supported
1056 };
1057
1058
1059 /** In correct XML the declaration is the first entry in the file.
1060     @verbatim
1061         <?xml version="1.0" standalone="yes"?>
1062     @endverbatim
1063
1064     TinyXML-2 will happily read or write files without a declaration,

```

```

1065     however.
1066
1067     The text of the declaration isn't interpreted. It is parsed
1068     and written as a string.
1069 */
1070 class TINYXML2_LIB XMLDeclaration : public XMLNode
1071 {
1072     friend class XMLDocument;
1073 public:
1074     virtual XMLDeclaration* ToDeclaration() {
1075         return this;
1076     }
1077     virtual const XMLDeclaration* ToDeclaration() const {
1078         return this;
1079     }
1080
1081     virtual bool Accept( XMLVisitor* visitor ) const;
1082
1083     virtual XMLNode* ShallowClone( XMLDocument* document ) const;
1084     virtual bool ShallowEqual( const XMLNode* compare ) const;
1085
1086 protected:
1087     explicit XMLDeclaration( XMLDocument* doc );
1088     virtual ~XMLDeclaration();
1089
1090     char* ParseDeep( char* p, StrPair* parentEndTag, int* curLineNumPtr );
1091
1092 private:
1093     XMLDeclaration( const XMLDeclaration& ); // not supported
1094     XMLDeclaration& operator=( const XMLDeclaration& ); // not supported
1095 };
1096
1097
1098 /** Any tag that TinyXML-2 doesn't recognize is saved as an
1099     unknown. It is a tag of text, but should not be modified.
1100     It will be written back to the XML, unchanged, when the file
1101     is saved.
1102
1103     DTD tags get thrown into XMLUnknowns.
1104 */
1105 class TINYXML2_LIB XMLUnknown : public XMLNode
1106 {
1107     friend class XMLDocument;
1108 public:
1109     virtual XMLUnknown* ToUnknown() {
1110         return this;
1111     }
1112     virtual const XMLUnknown* ToUnknown() const {
1113         return this;

```

```

1114 }
1115
1116 virtual bool Accept( XMLVisitor* visitor ) const;
1117
1118 virtual XMLNode* ShallowClone( XMLDocument* document ) const;
1119 virtual bool ShallowEqual( const XMLNode* compare ) const;
1120
1121 protected:
1122     explicit XMLUnknown( XMLDocument* doc );
1123     virtual ~XMLUnknown();
1124
1125     char* ParseDeep( char* p, StrPair* parentEndTag, int* curLineNumPtr );
1126
1127 private:
1128     XMLUnknown( const XMLUnknown& );    // not supported
1129     XMLUnknown& operator=( const XMLUnknown& ); // not supported
1130 };
1131
1132
1133
1134 /** An attribute is a name-value pair. Elements have an arbitrary
1135     number of attributes, each with a unique name.
1136
1137     @note The attributes are not XMLNodes. You may only query the
1138     Next() attribute in a list.
1139 */
1140 class TINYXML2_LIB XMLAttribute
1141 {
1142     friend class XMLElement;
1143 public:
1144     /// The name of the attribute.
1145     const char* Name() const;
1146
1147     /// The value of the attribute.
1148     const char* Value() const;
1149
1150     /// Gets the line number the attribute is in, if the document was parsed from a file.
1151     int GetLineNum() const { return _parseLineNum; }
1152
1153     /// The next attribute in the list.
1154     const XMLAttribute* Next() const {
1155         return _next;
1156     }
1157
1158     /** IntValue interprets the attribute as an integer, and returns the value.
1159         If the value isn't an integer, 0 will be returned. There is no error checking;
1160         use QueryIntValue() if you need error checking.
1161     */
1162     int IntValue() const {

```

```

1163         int i = 0;
1164         QueryIntValue(&i);
1165         return i;
1166     }
1167
1168     int64_t Int64Value() const {
1169         int64_t i = 0;
1170         QueryInt64Value(&i);
1171         return i;
1172     }
1173
1174     uint64_t Unsigned64Value() const {
1175         uint64_t i = 0;
1176         QueryUnsigned64Value(&i);
1177         return i;
1178     }
1179
1180     /// Query as an unsigned integer. See IntValue()
1181     unsigned UnsignedValue() const {
1182         unsigned i=0;
1183         QueryUnsignedValue( &i );
1184         return i;
1185     }
1186     /// Query as a boolean. See IntValue()
1187     bool BoolValue() const {
1188         bool b=false;
1189         QueryBoolValue( &b );
1190         return b;
1191     }
1192     /// Query as a double. See IntValue()
1193     double DoubleValue() const {
1194         double d=0;
1195         QueryDoubleValue( &d );
1196         return d;
1197     }
1198     /// Query as a float. See IntValue()
1199     float FloatValue() const {
1200         float f=0;
1201         QueryFloatValue( &f );
1202         return f;
1203     }
1204
1205     /** QueryIntValue interprets the attribute as an integer, and returns the value
1206         in the provided parameter. The function will return XML_SUCCESS on success,
1207         and XML_WRONG_ATTRIBUTE_TYPE if the conversion is not successful.
1208     */
1209     XMLError QueryIntValue( int* value ) const;
1210     /// See QueryIntValue
1211     XMLError QueryUnsignedValue( unsigned int* value ) const;

```

```

1212     /// See QueryIntValue
1213     XMLError QueryInt64Value(int64_t* value) const;
1214     /// See QueryIntValue
1215     XMLError QueryUnsigned64Value(uint64_t* value) const;
1216     /// See QueryIntValue
1217     XMLError QueryBoolValue( bool* value ) const;
1218     /// See QueryIntValue
1219     XMLError QueryDoubleValue( double* value ) const;
1220     /// See QueryIntValue
1221     XMLError QueryFloatValue( float* value ) const;
1222
1223     /// Set the attribute to a string value.
1224     void SetAttribute( const char* value );
1225     /// Set the attribute to value.
1226     void SetAttribute( int value );
1227     /// Set the attribute to value.
1228     void SetAttribute( unsigned value );
1229     /// Set the attribute to value.
1230     void SetAttribute(int64_t value);
1231     /// Set the attribute to value.
1232     void SetAttribute(uint64_t value);
1233     /// Set the attribute to value.
1234     void SetAttribute( bool value );
1235     /// Set the attribute to value.
1236     void SetAttribute( double value );
1237     /// Set the attribute to value.
1238     void SetAttribute( float value );
1239
1240 private:
1241     enum { BUF_SIZE = 200 };
1242
1243     XMLAttribute() : _name(), _value(), _parseLineNum( 0 ), _next( 0 ), _memPool( 0 ) {}
1244     virtual ~XMLAttribute()
1245     {}
1246
1247     XMLAttribute( const XMLAttribute& );    // not supported
1248     void operator=( const XMLAttribute& ); // not supported
1249     void SetName( const char* name );
1250
1251     char* ParseDeep( char* p, bool processEntities, int* curLineNumPtr );
1252
1253     mutable StrPair _name;
1254     mutable StrPair _value;
1255     int _parseLineNum;
1256     XMLAttribute* _next;
1257     MemPool* _memPool;
1258 };
1259

```

```

1260 /** The element is a container class. It has a value, the element name,
1261     and can contain other elements, text, comments, and unknowns.
1262     Elements also contain an arbitrary number of attributes.
1263 */
1264 class TINYXML2_LIB XMLElement : public XMLNode
1265 {
1266     friend class XMLDocument;
1267 public:
1268     /// Get the name of an element (which is the Value() of the node.)
1269     const char* Name() const    {
1270         return Value();
1271     }
1272     /// Set the name of the element.
1273     void SetName( const char* str, bool staticMem=false ) {
1274         SetValue( str, staticMem );
1275     }
1276
1277     virtual XMLElement* ToElement()                {
1278         return this;
1279     }
1280     virtual const XMLElement* ToElement() const {
1281         return this;
1282     }
1283     virtual bool Accept( XMLVisitor* visitor ) const;
1284
1285     /** Given an attribute name, Attribute() returns the value
1286         for the attribute of that name, or null if none
1287         exists. For example:
1288
1289         @verbatim
1290         const char* value = ele->Attribute( "foo" );
1291         @endverbatim
1292
1293         The 'value' parameter is normally null. However, if specified,
1294         the attribute will only be returned if the 'name' and 'value'
1295         match. This allow you to write code:
1296
1297         @verbatim
1298         if ( ele->Attribute( "foo", "bar" ) ) callFooIsBar();
1299         @endverbatim
1300
1301         rather than:
1302         @verbatim
1303         if ( ele->Attribute( "foo" ) ) {
1304             if ( strcmp( ele->Attribute( "foo" ), "bar" ) == 0 ) callFooIsBar();
1305         }
1306         @endverbatim
1307     */
1308     const char* Attribute( const char* name, const char* value=0 ) const;

```

```

1309
1310 /** Given an attribute name, IntAttribute() returns the value
1311     of the attribute interpreted as an integer. The default
1312     value will be returned if the attribute isn't present,
1313     or if there is an error. (For a method with error
1314     checking, see QueryIntAttribute()).
1315 */
1316     int IntAttribute(const char* name, int defaultValue = 0) const;
1317 /// See IntAttribute()
1318     unsigned UnsignedAttribute(const char* name, unsigned defaultValue = 0) const;
1319     /// See IntAttribute()
1320     int64_t Int64Attribute(const char* name, int64_t defaultValue = 0) const;
1321 /// See IntAttribute()
1322     uint64_t Unsigned64Attribute(const char* name, uint64_t defaultValue = 0) const;
1323     /// See IntAttribute()
1324     bool BoolAttribute(const char* name, bool defaultValue = false) const;
1325 /// See IntAttribute()
1326     double DoubleAttribute(const char* name, double defaultValue = 0) const;
1327 /// See IntAttribute()
1328     float FloatAttribute(const char* name, float defaultValue = 0) const;
1329
1330 /** Given an attribute name, QueryIntAttribute() returns
1331     XML_SUCCESS, XML_WRONG_ATTRIBUTE_TYPE if the conversion
1332     can't be performed, or XML_NO_ATTRIBUTE if the attribute
1333     doesn't exist. If successful, the result of the conversion
1334     will be written to 'value'. If not successful, nothing will
1335     be written to 'value'. This allows you to provide default
1336     value:
1337
1338     @verbatim
1339     int value = 10;
1340     QueryIntAttribute( "foo", &value );           // if "foo" isn't found, value will still be 10
1341     @endverbatim
1342 */
1343 XML_Error QueryIntAttribute( const char* name, int* value ) const {
1344     const XMLAttribute* a = FindAttribute( name );
1345     if ( !a ) {
1346         return XML_NO_ATTRIBUTE;
1347     }
1348     return a->QueryIntValue( value );
1349 }
1350
1351 /// See QueryIntAttribute()
1352 XML_Error QueryUnsignedAttribute( const char* name, unsigned int* value ) const {
1353     const XMLAttribute* a = FindAttribute( name );
1354     if ( !a ) {
1355         return XML_NO_ATTRIBUTE;
1356     }
1357     return a->QueryUnsignedValue( value );

```



```

1358 }
1359
1360 /// See QueryIntAttribute()
1361 XMLError QueryInt64Attribute(const char* name, int64_t* value) const {
1362     const XMLAttribute* a = FindAttribute(name);
1363     if (!a) {
1364         return XML_NO_ATTRIBUTE;
1365     }
1366     return a->QueryInt64Value(value);
1367 }
1368
1369 /// See QueryIntAttribute()
1370 XMLError QueryUnsigned64Attribute(const char* name, uint64_t* value) const {
1371     const XMLAttribute* a = FindAttribute(name);
1372     if (!a) {
1373         return XML_NO_ATTRIBUTE;
1374     }
1375     return a->QueryUnsigned64Value(value);
1376 }
1377
1378 /// See QueryIntAttribute()
1379 XMLError QueryBoolAttribute( const char* name, bool* value ) const {
1380     const XMLAttribute* a = FindAttribute( name );
1381     if ( !a ) {
1382         return XML_NO_ATTRIBUTE;
1383     }
1384     return a->QueryBoolValue( value );
1385 }
1386
1387 /// See QueryIntAttribute()
1388 XMLError QueryDoubleAttribute( const char* name, double* value ) const {
1389     const XMLAttribute* a = FindAttribute( name );
1390     if ( !a ) {
1391         return XML_NO_ATTRIBUTE;
1392     }
1393     return a->QueryDoubleValue( value );
1394 }
1395
1396 /// See QueryIntAttribute()
1397 XMLError QueryFloatAttribute( const char* name, float* value ) const {
1398     const XMLAttribute* a = FindAttribute( name );
1399     if ( !a ) {
1400         return XML_NO_ATTRIBUTE;
1401     }
1402     return a->QueryFloatValue( value );
1403 }
1404
1405 /// See QueryIntAttribute()
1406 XMLError QueryStringAttribute(const char* name, const char** value) const {
1407     const XMLAttribute* a = FindAttribute(name);
1408     if (!a) {

```

```

1407         return XML_NO_ATTRIBUTE;
1408     }
1409     *value = a->Value();
1410     return XML_SUCCESS;
1411 }
1412
1413
1414
1415 /** Given an attribute name, QueryAttribute() returns
1416     XML_SUCCESS, XML_WRONG_ATTRIBUTE_TYPE if the conversion
1417     can't be performed, or XML_NO_ATTRIBUTE if the attribute
1418     doesn't exist. It is overloaded for the primitive types,
1419     and is a generally more convenient replacement of
1420     QueryIntAttribute() and related functions.
1421
1422     If successful, the result of the conversion
1423     will be written to 'value'. If not successful, nothing will
1424     be written to 'value'. This allows you to provide default
1425     value:
1426
1427     @verbatim
1428     int value = 10;
1429     QueryAttribute( "foo", &value );           // if "foo" isn't found, value will still be 10
1430     @endverbatim
1431 */
1432 XML_Error QueryAttribute( const char* name, int* value ) const {
1433     return QueryIntAttribute( name, value );
1434 }
1435
1436 XML_Error QueryAttribute( const char* name, unsigned int* value ) const {
1437     return QueryUnsignedAttribute( name, value );
1438 }
1439
1440 XML_Error QueryAttribute(const char* name, int64_t* value) const {
1441     return QueryInt64Attribute(name, value);
1442 }
1443
1444 XML_Error QueryAttribute(const char* name, uint64_t* value) const {
1445     return QueryUnsigned64Attribute(name, value);
1446 }
1447
1448 XML_Error QueryAttribute( const char* name, bool* value ) const {
1449     return QueryBoolAttribute( name, value );
1450 }
1451
1452 XML_Error QueryAttribute( const char* name, double* value ) const {
1453     return QueryDoubleAttribute( name, value );
1454 }
1455

```

```

1456 XMLError QueryAttribute( const char* name, float* value ) const {
1457     return QueryFloatAttribute( name, value );
1458 }
1459
1460 XMLError QueryAttribute(const char* name, const char** value) const {
1461     return QueryStringAttribute(name, value);
1462 }
1463
1464 /// Sets the named attribute to value.
1465 void SetAttribute( const char* name, const char* value )    {
1466     XMLAttribute* a = FindOrCreateAttribute( name );
1467     a->SetAttribute( value );
1468 }
1469 /// Sets the named attribute to value.
1470 void SetAttribute( const char* name, int value )            {
1471     XMLAttribute* a = FindOrCreateAttribute( name );
1472     a->SetAttribute( value );
1473 }
1474 /// Sets the named attribute to value.
1475 void SetAttribute( const char* name, unsigned value )      {
1476     XMLAttribute* a = FindOrCreateAttribute( name );
1477     a->SetAttribute( value );
1478 }
1479
1480 /// Sets the named attribute to value.
1481 void SetAttribute(const char* name, int64_t value) {
1482     XMLAttribute* a = FindOrCreateAttribute(name);
1483     a->SetAttribute(value);
1484 }
1485
1486 /// Sets the named attribute to value.
1487 void SetAttribute(const char* name, uint64_t value) {
1488     XMLAttribute* a = FindOrCreateAttribute(name);
1489     a->SetAttribute(value);
1490 }
1491
1492 /// Sets the named attribute to value.
1493 void SetAttribute( const char* name, bool value )          {
1494     XMLAttribute* a = FindOrCreateAttribute( name );
1495     a->SetAttribute( value );
1496 }
1497 /// Sets the named attribute to value.
1498 void SetAttribute( const char* name, double value )        {
1499     XMLAttribute* a = FindOrCreateAttribute( name );
1500     a->SetAttribute( value );
1501 }
1502 /// Sets the named attribute to value.
1503 void SetAttribute( const char* name, float value )         {
1504     XMLAttribute* a = FindOrCreateAttribute( name );

```

```

1505     a->SetAttribute( value );
1506 }
1507
1508 /**
1509     Delete an attribute.
1510 */
1511 void DeleteAttribute( const char* name );
1512
1513 /// Return the first attribute in the list.
1514 const XMLAttribute* FirstAttribute() const {
1515     return _rootAttribute;
1516 }
1517 /// Query a specific attribute in the list.
1518 const XMLAttribute* FindAttribute( const char* name ) const;
1519
1520 /** Convenience function for easy access to the text inside an element. Although easy
1521     and concise, GetText() is limited compared to getting the XMLText child
1522     and accessing it directly.
1523
1524     If the first child of 'this' is a XMLText, the GetText()
1525     returns the character string of the Text node, else null is returned.
1526
1527     This is a convenient method for getting the text of simple contained text:
1528     @verbatim
1529     <foo>This is text</foo>
1530         const char* str = fooElement->GetText();
1531     @endverbatim
1532
1533     'str' will be a pointer to "This is text".
1534
1535     Note that this function can be misleading. If the element foo was created from
1536     this XML:
1537     @verbatim
1538         <foo><b>This is text</b></foo>
1539     @endverbatim
1540
1541     then the value of str would be null. The first child node isn't a text node, it is
1542     another element. From this XML:
1543     @verbatim
1544         <foo>This is <b>text</b></foo>
1545     @endverbatim
1546     GetText() will return "This is ".
1547 */
1548 const char* GetText() const;
1549
1550 /** Convenience function for easy access to the text inside an element. Although easy
1551     and concise, SetText() is limited compared to creating an XMLText child
1552     and mutating it directly.
1553

```

If the first child of 'this' is a XMLText, SetText() sets its value to the given string, otherwise it will create a first child that is an XMLText.

This is a convenient method for setting the text of simple contained text:

```
@verbatim
<foo>This is text</foo>
    fooElement->SetText( "Hullaballoo!" );
<foo>Hullaballoo!</foo>
@endverbatim
```

Note that this function can be misleading. If the element foo was created from this XML:

```
@verbatim
    <foo><b>This is text</b></foo>
@endverbatim
```

then it will not change "This is text", but rather prefix it with a text element:

```
@verbatim
    <foo>Hullaballoo!<b>This is text</b></foo>
@endverbatim
```

For this XML:

```
@verbatim
    <foo />
@endverbatim
SetText() will generate
@verbatim
    <foo>Hullaballoo!</foo>
@endverbatim
```

*/

```
void SetText( const char* inText );
```

/// Convenience method for setting text inside an element. See SetText() for important limitations.

```
void SetText( int value );
```

/// Convenience method for setting text inside an element. See SetText() for important limitations.

```
void SetText( unsigned value );
```

/// Convenience method for setting text inside an element. See SetText() for important limitations.

```
void SetText(int64_t value);
```

/// Convenience method for setting text inside an element. See SetText() for important limitations.

```
void SetText(uint64_t value);
```

/// Convenience method for setting text inside an element. See SetText() for important limitations.

```
void SetText( bool value );
```

/// Convenience method for setting text inside an element. See SetText() for important limitations.

```
void SetText( double value );
```

/// Convenience method for setting text inside an element. See SetText() for important limitations.

```
void SetText( float value );
```

/**

Convenience method to query the value of a child text node. This is probably best shown by example. Given you have a document is this form:

```

1603     @verbatim
1604         <point>
1605             <x>1</x>
1606             <y>1.4</y>
1607         </point>
1608     @endverbatim
1609
1610     The QueryIntText() and similar functions provide a safe and easier way to get to the
1611     "value" of x and y.
1612
1613     @verbatim
1614         int x = 0;
1615         float y = 0;    // types of x and y are contrived for example
1616         const XMLElement* xElement = pointElement->FirstChildElement( "x" );
1617         const XMLElement* yElement = pointElement->FirstChildElement( "y" );
1618         xElement->QueryIntText( &x );
1619         yElement->QueryFloatText( &y );
1620     @endverbatim
1621
1622     @returns XML_SUCCESS (0) on success, XML_CAN_NOT_CONVERT_TEXT if the text cannot be
converted
1623         to the requested type, and XML_NO_TEXT_NODE if there is no child text to query.
1624
1625     */
1626     XMLError QueryIntText( int* ival ) const;
1627     /// See QueryIntText()
1628     XMLError QueryUnsignedText( unsigned* uval ) const;
1629     /// See QueryIntText()
1630     XMLError QueryInt64Text( int64_t* uval ) const;
1631     /// See QueryIntText()
1632     XMLError QueryUnsigned64Text( uint64_t* uval ) const;
1633     /// See QueryIntText()
1634     XMLError QueryBoolText( bool* bval ) const;
1635     /// See QueryIntText()
1636     XMLError QueryDoubleText( double* dval ) const;
1637     /// See QueryIntText()
1638     XMLError QueryFloatText( float* fval ) const;
1639
1640     int IntText( int defaultValue = 0 ) const;
1641
1642     /// See QueryIntText()
1643     unsigned UnsignedText( unsigned defaultValue = 0 ) const;
1644     /// See QueryIntText()
1645     int64_t Int64Text( int64_t defaultValue = 0 ) const;
1646     /// See QueryIntText()
1647     uint64_t Unsigned64Text( uint64_t defaultValue = 0 ) const;
1648     /// See QueryIntText()
1649     bool BoolText( bool defaultValue = false ) const;
1650     /// See QueryIntText()

```

```

1651     double DoubleText(double defaultValue = 0) const;
1652     /// See QueryIntText()
1653     float FloatText(float defaultValue = 0) const;
1654
1655     /**
1656      Convenience method to create a new XElement and add it as last (right)
1657      child of this node. Returns the created and inserted element.
1658     */
1659     XElement* InsertNewChildElement(const char* name);
1660     /// See InsertNewChildElement()
1661     XMLComment* InsertNewComment(const char* comment);
1662     /// See InsertNewChildElement()
1663     XMLText* InsertNewText(const char* text);
1664     /// See InsertNewChildElement()
1665     XMLDeclaration* InsertNewDeclaration(const char* text);
1666     /// See InsertNewChildElement()
1667     XMLUnknown* InsertNewUnknown(const char* text);
1668
1669
1670     // internal:
1671     enum ElementClosingType {
1672         OPEN,          // <foo>
1673         CLOSED,        // <foo/>
1674         CLOSING         // </foo>
1675     };
1676     ElementClosingType ClosingType() const {
1677         return _closingType;
1678     }
1679     virtual XMLNode* ShallowClone( XMLDocument* document ) const;
1680     virtual bool ShallowEqual( const XMLNode* compare ) const;
1681
1682 protected:
1683     char* ParseDeep( char* p, StrPair* parentEndTag, int* curLineNumPtr );
1684
1685 private:
1686     XElement( XMLDocument* doc );
1687     virtual ~XMLElement();
1688     XElement( const XElement& ); // not supported
1689     void operator=( const XElement& ); // not supported
1690
1691     XMLAttribute* FindOrCreateAttribute( const char* name );
1692     char* ParseAttributes( char* p, int* curLineNumPtr );
1693     static void DeleteAttribute( XMLAttribute* attribute );
1694     XMLAttribute* CreateAttribute();
1695
1696     enum { BUF_SIZE = 200 };
1697     ElementClosingType _closingType;
1698     // The attribute list is ordered; there is no 'lastAttribute'
1699     // because the list needs to be scanned for dupes before adding

```

```

1700 // a new attribute.
1701 XMLAttribute* _rootAttribute;
1702 };
1703
1704
1705 enum Whitespace {
1706     PRESERVE_WHITESPACE,
1707     COLLAPSE_WHITESPACE
1708 };
1709
1710
1711 /** A Document binds together all the functionality.
1712     It can be saved, loaded, and printed to the screen.
1713     All Nodes are connected and allocated to a Document.
1714     If the Document is deleted, all its Nodes are also deleted.
1715 */
1716 class TINYXML2_LIB XMLDocument : public XMLNode
1717 {
1718     friend class XMLElement;
1719     // Gives access to SetError and Push/PopDepth, but over-access for everything else.
1720     // Wishing C++ had "internal" scope.
1721     friend class XMLNode;
1722     friend class XMLText;
1723     friend class XMLComment;
1724     friend class XMLDeclaration;
1725     friend class XMLUnknown;
1726 public:
1727     /// constructor
1728     XMLDocument( bool processEntities = true, Whitespace whitespaceMode = PRESERVE_WHITESPACE );
1729     ~XMLDocument();
1730
1731     virtual XMLDocument* ToDocument() {
1732         TIXMLASSERT( this == _document );
1733         return this;
1734     }
1735     virtual const XMLDocument* ToDocument() const {
1736         TIXMLASSERT( this == _document );
1737         return this;
1738     }
1739
1740 /**
1741     Parse an XML file from a character string.
1742     Returns XML_SUCCESS (0) on success, or
1743     an errorID.
1744
1745     You may optionally pass in the 'nBytes', which is
1746     the number of bytes which will be parsed. If not
1747     specified, TinyXML-2 will assume 'xml' points to a
1748     null terminated string.

```



```

1749 */
1750 XML_Error Parse( const char* xml, size_t nBytes=static_cast<size_t>(-1) );
1751
1752 /**
1753     Load an XML file from disk.
1754     Returns XML_SUCCESS (0) on success, or
1755     an errorID.
1756 */
1757 XML_Error LoadFile( const char* filename );
1758
1759 /**
1760     Load an XML file from disk. You are responsible
1761     for providing and closing the FILE*.
1762
1763     NOTE: The file should be opened as binary ("rb")
1764     not text in order for TinyXML-2 to correctly
1765     do newline normalization.
1766
1767     Returns XML_SUCCESS (0) on success, or
1768     an errorID.
1769 */
1770 XML_Error LoadFile( FILE* );
1771
1772 /**
1773     Save the XML file to disk.
1774     Returns XML_SUCCESS (0) on success, or
1775     an errorID.
1776 */
1777 XML_Error SaveFile( const char* filename, bool compact = false );
1778
1779 /**
1780     Save the XML file to disk. You are responsible
1781     for providing and closing the FILE*.
1782
1783     Returns XML_SUCCESS (0) on success, or
1784     an errorID.
1785 */
1786 XML_Error SaveFile( FILE* fp, bool compact = false );
1787
1788 bool ProcessEntities() const {
1789     return _processEntities;
1790 }
1791 Whitespace WhitespaceMode() const {
1792     return _whitespaceMode;
1793 }
1794
1795 /**
1796     Returns true if this document has a leading Byte Order Mark of UTF8.
1797 */

```

```

1798 bool HasBOM() const {
1799     return _writeBOM;
1800 }
1801 /** Sets whether to write the BOM when writing the file.
1802 */
1803 void SetBOM( bool useBOM ) {
1804     _writeBOM = useBOM;
1805 }
1806
1807 /** Return the root element of DOM. Equivalent to FirstChildElement().
1808     To get the first node, use FirstChild().
1809 */
1810 XMLElement* RootElement() {
1811     return FirstChildElement();
1812 }
1813 const XMLElement* RootElement() const {
1814     return FirstChildElement();
1815 }
1816
1817 /** Print the Document. If the Printer is not provided, it will
1818     print to stdout. If you provide Printer, this can print to a file:
1819     @verbatim
1820     XMLPrinter printer( fp );
1821     doc.Print( &printer );
1822     @endverbatim
1823
1824     Or you can use a printer to print to memory:
1825     @verbatim
1826     XMLPrinter printer;
1827     doc.Print( &printer );
1828     // printer.CStr() has a const char* to the XML
1829     @endverbatim
1830 */
1831 void Print( XMLPrinter* streamer=0 ) const;
1832 virtual bool Accept( XMLVisitor* visitor ) const;
1833
1834 /**
1835     Create a new Element associated with
1836     this Document. The memory for the Element
1837     is managed by the Document.
1838 */
1839 XMLElement* NewElement( const char* name );
1840 /**
1841     Create a new Comment associated with
1842     this Document. The memory for the Comment
1843     is managed by the Document.
1844 */
1845 XMLComment* NewComment( const char* comment );
1846 /**

```

```
1847     Create a new Text associated with
1848     this Document. The memory for the Text
1849     is managed by the Document.
1850 */
1851 XMLText* NewText( const char* text );
1852 /**
1853     Create a new Declaration associated with
1854     this Document. The memory for the object
1855     is managed by the Document.
1856
1857     If the 'text' param is null, the standard
1858     declaration is used.:
1859     @verbatim
1860         <?xml version="1.0" encoding="UTF-8"?>
1861     @endverbatim
1862 */
1863 XMLDeclaration* NewDeclaration( const char* text=0 );
1864 /**
1865     Create a new Unknown associated with
1866     this Document. The memory for the object
1867     is managed by the Document.
1868 */
1869 XMLUnknown* NewUnknown( const char* text );
1870
1871 /**
1872     Delete a node associated with this document.
1873     It will be unlinked from the DOM.
1874 */
1875 void DeleteNode( XMLNode* node );
1876
1877 /// Clears the error flags.
1878 void ClearError();
1879
1880 /// Return true if there was an error parsing the document.
1881 bool Error() const {
1882     return _errorID != XML_SUCCESS;
1883 }
1884 /// Return the errorID.
1885 XMLError ErrorID() const {
1886     return _errorID;
1887 }
1888     const char* ErrorName() const;
1889 static const char* ErrorIDToName(XMLError errorID);
1890
1891 /** Returns a "long form" error description. A hopefully helpful
1892     diagnostic with location, line number, and/or additional info.
1893 */
1894     const char* ErrorStr() const;
1895
```

```

1896 /// A (trivial) utility function that prints the ErrorStr() to stdout.
1897 void PrintError() const;
1898
1899 /// Return the line where the error occurred, or zero if unknown.
1900 int ErrorLineNum() const
1901 {
1902     return _errorLineNum;
1903 }
1904
1905 /// Clear the document, resetting it to the initial state.
1906 void Clear();
1907
1908 /**
1909     Copies this document to a target document.
1910     The target will be completely cleared before the copy.
1911     If you want to copy a sub-tree, see XMLNode::DeepClone().
1912
1913     NOTE: that the 'target' must be non-null.
1914 */
1915 void DeepCopy(XMLDocument* target) const;
1916
1917 // internal
1918 char* Identify( char* p, XMLNode** node );
1919
1920 // internal
1921 void MarkInUse(const XMLNode* const);
1922
1923 virtual XMLNode* ShallowClone( XMLDocument* /*document*/ ) const {
1924     return 0;
1925 }
1926 virtual bool ShallowEqual( const XMLNode* /*compare*/ ) const {
1927     return false;
1928 }
1929
1930 private:
1931 XMLDocument( const XMLDocument& ); // not supported
1932 void operator=( const XMLDocument& ); // not supported
1933
1934 bool _writeBOM;
1935 bool
1936 _processEntities;
1937 XMLError _errorID;
1938 Whitespace _whitespaceMode;
1939 mutable StrPair _errorStr;
1940 int _errorLineNum;
1941 char*
1942 _charBuffer;
1943 int
1944 _parseCurLineNum;

```

```

1942     int                _parsingDepth;
1943     // Memory tracking does add some overhead.
1944     // However, the code assumes that you don't
1945     // have a bunch of unlinked nodes around.
1946     // Therefore it takes less memory to track
1947     // in the document vs. a linked list in the XMLNode,
1948     // and the performance is the same.
1949     DynArray<XMLNode*, 10> _unlinked;
1950
1951     MemPoolT< sizeof(XMLElement) >   _elementPool;
1952     MemPoolT< sizeof(XMLAttribute) > _attributePool;
1953     MemPoolT< sizeof(XMLText) >      _textPool;
1954     MemPoolT< sizeof(XMLComment) >   _commentPool;
1955
1956     static const char* _errorNames[XML_ERROR_COUNT];
1957
1958     void Parse();
1959
1960     void SetError( XMLError error, int lineNum, const char* format, ... );
1961
1962     // Something of an obvious security hole, once it was discovered.
1963     // Either an ill-formed XML or an excessively deep one can overflow
1964     // the stack. Track stack depth, and error out if needed.
1965     class DepthTracker {
1966     public:
1967         explicit DepthTracker(XMLDocument * document) {
1968             this->_document = document;
1969             document->PushDepth();
1970         }
1971         ~DepthTracker() {
1972             _document->PopDepth();
1973         }
1974     private:
1975         XMLDocument * _document;
1976     };
1977     void PushDepth();
1978     void PopDepth();
1979
1980     template<class NodeType, int PoolElementSize>
1981     NodeType* CreateUnlinkedNode( MemPoolT<PoolElementSize>& pool );
1982 };
1983
1984 template<class NodeType, int PoolElementSize>
1985 inline NodeType* XMLDocument::CreateUnlinkedNode( MemPoolT<PoolElementSize>& pool )
1986 {
1987     TIXMLASSERT( sizeof( NodeType ) == PoolElementSize );
1988     TIXMLASSERT( sizeof( NodeType ) == pool.ItemSize() );
1989     NodeType* returnNode = new (pool.Alloc()) NodeType( this );
1990     TIXMLASSERT( returnNode );

```

```

1991     returnNode->_memPool = &pool;
1992
1993     _unlinked.Push(returnNode);
1994     return returnNode;
1995 }
1996
1997 /**
1998     A XMLHandle is a class that wraps a node pointer with null checks; this is
1999     an incredibly useful thing. Note that XMLHandle is not part of the TinyXML-2
2000     DOM structure. It is a separate utility class.
2001
2002     Take an example:
2003     @verbatim
2004     <Document>
2005         <Element attributeA = "valueA">
2006             <Child attributeB = "value1" />
2007             <Child attributeB = "value2" />
2008         </Element>
2009     </Document>
2010     @endverbatim
2011
2012     Assuming you want the value of "attributeB" in the 2nd "Child" element, it's very
2013     easy to write a lot of code that looks like:
2014
2015     @verbatim
2016     XMLElement* root = document.FirstChildElement( "Document" );
2017     if ( root )
2018     {
2019         XMLElement* element = root->FirstChildElement( "Element" );
2020         if ( element )
2021         {
2022             XMLElement* child = element->FirstChildElement( "Child" );
2023             if ( child )
2024             {
2025                 XMLElement* child2 = child->NextSiblingElement( "Child" );
2026                 if ( child2 )
2027                 {
2028                     // Finally do something useful.
2029
2030                 @endverbatim
2031
2032     And that doesn't even cover "else" cases. XMLHandle addresses the verbosity
2033     of such code. A XMLHandle checks for null pointers so it is perfectly safe
2034     and correct to use:
2035
2036     @verbatim
2037     XMLHandle docHandle( &document );
2038     XMLElement* child2 = docHandle.FirstChildElement( "Document" ).FirstChildElement( "Element"
    ).FirstChildElement().NextSiblingElement();
    if ( child2 )

```

```

2039     {
2040         // do something useful
2041     @endverbatim
2042
2043     Which is MUCH more concise and useful.
2044
2045     It is also safe to copy handles - internally they are nothing more than node pointers.
2046     @verbatim
2047     XMLHandle handleCopy = handle;
2048     @endverbatim
2049
2050     See also XMLConstHandle, which is the same as XMLHandle, but operates on const objects.
2051 */
2052 class TINYXML2_LIB XMLHandle
2053 {
2054 public:
2055     /// Create a handle from any node (at any depth of the tree.) This can be a null pointer.
2056     explicit XMLHandle( XMLNode* node ) : _node( node ) {
2057     }
2058     /// Create a handle from a node.
2059     explicit XMLHandle( XMLNode& node ) : _node( &node ) {
2060     }
2061     /// Copy constructor
2062     XMLHandle( const XMLHandle& ref ) : _node( ref._node ) {
2063     }
2064     /// Assignment
2065     XMLHandle& operator=( const XMLHandle& ref )
2066     {
2067         _node = ref._node;
2068         return *this;
2069     }
2070     /// Get the first child of this handle.
2071     XMLHandle FirstChild() {
2072         return XMLHandle( _node ? _node->FirstChild() : 0 );
2073     }
2074     /// Get the first child element of this handle.
2075     XMLHandle FirstChildElement( const char* name = 0 ) {
2076         return XMLHandle( _node ? _node->FirstChildElement( name ) : 0 );
2077     }
2078     /// Get the last child of this handle.
2079     XMLHandle LastChild() {
2080         return XMLHandle( _node ? _node->LastChild() : 0 );
2081     }
2082     /// Get the last child element of this handle.
2083     XMLHandle LastChildElement( const char* name = 0 ) {
2084         return XMLHandle( _node ? _node->LastChildElement( name ) : 0 );
2085     }
2086     /// Get the previous sibling of this handle.

```

```

2087 XMLHandle PreviousSibling()                                {
2088     return XMLHandle( _node ? _node->PreviousSibling() : 0 );
2089 }
2090 /// Get the previous sibling element of this handle.
2091 XMLHandle PreviousSiblingElement( const char* name = 0 )    {
2092     return XMLHandle( _node ? _node->PreviousSiblingElement( name ) : 0 );
2093 }
2094 /// Get the next sibling of this handle.
2095 XMLHandle NextSibling()                                    {
2096     return XMLHandle( _node ? _node->NextSibling() : 0 );
2097 }
2098 /// Get the next sibling element of this handle.
2099 XMLHandle NextSiblingElement( const char* name = 0 )        {
2100     return XMLHandle( _node ? _node->NextSiblingElement( name ) : 0 );
2101 }
2102
2103 /// Safe cast to XMLNode. This can return null.
2104 XMLNode* ToNode()                                          {
2105     return _node;
2106 }
2107 /// Safe cast to XMLElement. This can return null.
2108 XMLElement* ToElement()                                  {
2109     return ( _node ? _node->ToElement() : 0 );
2110 }
2111 /// Safe cast to XMLText. This can return null.
2112 XMLText* ToText()                                         {
2113     return ( _node ? _node->ToText() : 0 );
2114 }
2115 /// Safe cast to XMLUnknown. This can return null.
2116 XMLUnknown* ToUnknown()                                   {
2117     return ( _node ? _node->ToUnknown() : 0 );
2118 }
2119 /// Safe cast to XMLDeclaration. This can return null.
2120 XMLDeclaration* ToDeclaration()                           {
2121     return ( _node ? _node->ToDeclaration() : 0 );
2122 }
2123
2124 private:
2125     XMLNode* _node;
2126 };
2127
2128 /**
2129     A variant of the XMLHandle class for working with const XMLNodes and Documents. It is the
2130     same in all regards, except for the 'const' qualifiers. See XMLHandle for API.
2131 */
2132 */
2133 class TINYXML2_LIB XMLConstHandle
2134 {
2135 public:

```



```

2136 explicit XMLConstHandle( const XMLNode* node ) : _node( node ) {
2137 }
2138 explicit XMLConstHandle( const XMLNode& node ) : _node( &node ) {
2139 }
2140 XMLConstHandle( const XMLConstHandle& ref ) : _node( ref._node ) {
2141 }
2142
2143 XMLConstHandle& operator=( const XMLConstHandle& ref ) {
2144     _node = ref._node;
2145     return *this;
2146 }
2147
2148 const XMLConstHandle FirstChild() const {
2149     return XMLConstHandle( _node ? _node->FirstChild() : 0 );
2150 }
2151 const XMLConstHandle FirstChildElement( const char* name = 0 ) const {
2152     return XMLConstHandle( _node ? _node->FirstChildElement( name ) : 0 );
2153 }
2154 const XMLConstHandle LastChild() const {
2155     return XMLConstHandle( _node ? _node->LastChild() : 0 );
2156 }
2157 const XMLConstHandle LastChildElement( const char* name = 0 ) const {
2158     return XMLConstHandle( _node ? _node->LastChildElement( name ) : 0 );
2159 }
2160 const XMLConstHandle PreviousSibling() const {
2161     return XMLConstHandle( _node ? _node->PreviousSibling() : 0 );
2162 }
2163 const XMLConstHandle PreviousSiblingElement( const char* name = 0 ) const {
2164     return XMLConstHandle( _node ? _node->PreviousSiblingElement( name ) : 0 );
2165 }
2166 const XMLConstHandle NextSibling() const {
2167     return XMLConstHandle( _node ? _node->NextSibling() : 0 );
2168 }
2169 const XMLConstHandle NextSiblingElement( const char* name = 0 ) const {
2170     return XMLConstHandle( _node ? _node->NextSiblingElement( name ) : 0 );
2171 }
2172
2173
2174 const XMLNode* ToNode() const {
2175     return _node;
2176 }
2177 const XMLElement* ToElement() const {
2178     return ( _node ? _node->ToElement() : 0 );
2179 }
2180 const XMLText* ToText() const {
2181     return ( _node ? _node->ToText() : 0 );
2182 }
2183 const XMLUnknown* ToUnknown() const {
2184     return ( _node ? _node->ToUnknown() : 0 );

```

```

2185 }
2186 const XMLDeclaration* ToDeclaration() const {
2187     return ( _node ? _node->ToDeclaration() : 0 );
2188 }
2189
2190 private:
2191     const XMLNode* _node;
2192 };
2193
2194
2195 /**
2196     Printing functionality. The XMLPrinter gives you more
2197     options than the XMLDocument::Print() method.
2198
2199     It can:
2200     -# Print to memory.
2201     -# Print to a file you provide.
2202     -# Print XML without a XMLDocument.
2203
2204     Print to Memory
2205
2206     @verbatim
2207     XMLPrinter printer;
2208     doc.Print( &printer );
2209     SomeFunction( printer.CStr() );
2210     @endverbatim
2211
2212     Print to a File
2213
2214     You provide the file pointer.
2215     @verbatim
2216     XMLPrinter printer( fp );
2217     doc.Print( &printer );
2218     @endverbatim
2219
2220     Print without a XMLDocument
2221
2222     When loading, an XML parser is very useful. However, sometimes
2223     when saving, it just gets in the way. The code is often set up
2224     for streaming, and constructing the DOM is just overhead.
2225
2226     The Printer supports the streaming case. The following code
2227     prints out a trivially simple XML file without ever creating
2228     an XML document.
2229
2230     @verbatim
2231     XMLPrinter printer( fp );
2232     printer.OpenElement( "foo" );
2233     printer.PushAttribute( "foo", "bar" );

```

```

2234     printer.CloseElement();
2235     @endverbatim
2236 */
2237 class TINYXML2_LIB XMLPrinter : public XMLVisitor
2238 {
2239 public:
2240     /** Construct the printer. If the FILE* is specified,
2241         this will print to the FILE. Else it will print
2242         to memory, and the result is available in CStr().
2243         If 'compact' is set to true, then output is created
2244         with only required whitespace and newlines.
2245     */
2246     XMLPrinter( FILE* file=0, bool compact = false, int depth = 0 );
2247     virtual ~XMLPrinter() {}
2248
2249     /** If streaming, write the BOM and declaration. */
2250     void PushHeader( bool writeBOM, bool writeDeclaration );
2251     /** If streaming, start writing an element.
2252         The element must be closed with CloseElement()
2253     */
2254     void OpenElement( const char* name, bool compactMode=false );
2255     /// If streaming, add an attribute to an open element.
2256     void PushAttribute( const char* name, const char* value );
2257     void PushAttribute( const char* name, int value );
2258     void PushAttribute( const char* name, unsigned value );
2259     void PushAttribute( const char* name, int64_t value );
2260     void PushAttribute( const char* name, uint64_t value );
2261     void PushAttribute( const char* name, bool value );
2262     void PushAttribute( const char* name, double value );
2263     /// If streaming, close the Element.
2264     virtual void CloseElement( bool compactMode=false );
2265
2266     /// Add a text node.
2267     void PushText( const char* text, bool cdata=false );
2268     /// Add a text node from an integer.
2269     void PushText( int value );
2270     /// Add a text node from an unsigned.
2271     void PushText( unsigned value );
2272     /// Add a text node from a signed 64bit integer.
2273     void PushText( int64_t value );
2274     /// Add a text node from an unsigned 64bit integer.
2275     void PushText( uint64_t value );
2276     /// Add a text node from a bool.
2277     void PushText( bool value );
2278     /// Add a text node from a float.
2279     void PushText( float value );
2280     /// Add a text node from a double.
2281     void PushText( double value );
2282

```

```

2283 /// Add a comment
2284 void PushComment( const char* comment );
2285
2286 void PushDeclaration( const char* value );
2287 void PushUnknown( const char* value );
2288
2289 virtual bool VisitEnter( const XMLDocument& /*doc*/ );
2290 virtual bool VisitExit( const XMLDocument& /*doc*/ )           {
2291     return true;
2292 }
2293
2294 virtual bool VisitEnter( const XMLElement& element, const XMLAttribute* attribute );
2295 virtual bool VisitExit( const XMLElement& element );
2296
2297 virtual bool Visit( const XMLText& text );
2298 virtual bool Visit( const XMLComment& comment );
2299 virtual bool Visit( const XMLDeclaration& declaration );
2300 virtual bool Visit( const XMLUnknown& unknown );
2301
2302 /**
2303     If in print to memory mode, return a pointer to
2304     the XML file in memory.
2305 */
2306 const char* CStr() const {
2307     return _buffer.Mem();
2308 }
2309 /**
2310     If in print to memory mode, return the size
2311     of the XML file in memory. (Note the size returned
2312     includes the terminating null.)
2313 */
2314 int CStrSize() const {
2315     return _buffer.Size();
2316 }
2317 /**
2318     If in print to memory mode, reset the buffer to the
2319     beginning.
2320 */
2321 void ClearBuffer( bool resetToFirstElement = true ) {
2322     _buffer.Clear();
2323     _buffer.Push(0);
2324     _firstElement = resetToFirstElement;
2325 }
2326
2327 protected:
2328     virtual bool CompactMode( const XMLElement& ) { return _compactMode; }
2329
2330     /** Prints out the space before an element. You may override to change
2331         the space and tabs used. A PrintSpace() override should call Print().

```

```

2332     */
2333     virtual void PrintSpace( int depth );
2334     virtual void Print( const char* format, ... );
2335     virtual void Write( const char* data, size_t size );
2336     virtual void Putc( char ch );
2337
2338     inline void Write(const char* data) { Write(data, strlen(data)); }
2339
2340     void SealElementIfJustOpened();
2341     bool _elementJustOpened;
2342     DynArray< const char*, 10 > _stack;
2343
2344 private:
2345     /**
2346         Prepares to write a new node. This includes sealing an element that was
2347         just opened, and writing any whitespace necessary if not in compact mode.
2348     */
2349     void PrepareForNewNode( bool compactMode );
2350     void PrintString( const char*, bool restrictedEntitySet );    // prints out, after detecting entities.
2351
2352     bool _firstElement;
2353     FILE* _fp;
2354     int _depth;
2355     int _textDepth;
2356     bool _processEntities;
2357     bool _compactMode;
2358
2359     enum {
2360         ENTITY_RANGE = 64,
2361         BUF_SIZE = 200
2362     };
2363     bool _entityFlag[ENTITY_RANGE];
2364     bool _restrictedEntityFlag[ENTITY_RANGE];
2365
2366     DynArray< char, 20 > _buffer;
2367
2368     // Prohibit cloning, intentionally not implemented
2369     XMLPrinter( const XMLPrinter& );
2370     XMLPrinter& operator=( const XMLPrinter& );
2371 };
2372
2373
2374 }    // tinyxml2
2375
2376 #if defined(_MSC_VER)
2377 # pragma warning(pop)
2378 #endif
2379
2380 #endif // TINYXML2_INCLUDED

```

