CEC Intermediate Representation

Stephen A. Edwards
Columbia University
sedwards@cs.columbia.edu

Abstract

This package provides a lightweight mechanism for making C++ objects persistent. Each persistent object must derive from the `IR::Node` class and implement simple `read` and `write` member functions that fill in and write its fields to XML input and output streams.

The XML reading facility is built on top of James Clark's standard *expat* XML parser, and first builds an in-memory tree of the XML document before traversing it to build a tree of `IR::Node`-derived objects.

Self-referential (cyclic) data structures are supported by writing each object at most once and including references in the XML file where necessary.

Contents

1 IR Nodes

1.1 The Node Class

All nodes in the IR are derived from `IR::Node`. Each such class has a single static member called `_` that holds its name, which the `className` method returns. The XML writing methods use this method to obtain the element name for each object.

The `XMLostream` class uses the `write` method to send the contents of the object's fields to the XML output stream. Similarly, `XMListream` fills the object's fields with data from an XML input stream.

2a ⟨*Node class* 2a⟩≡ (17b)

```
class Node {
  friend class XMLostream;
  friend class XMListream;
protected:
  virtual void read(XMListream &) {}
  virtual void write(XMLostream &) const {}
public:
  static IR::Class _;
  virtual const std::string className() const { return _; }
  virtual ~Node() {}
};
```

Each class in the IR must register itself with the `Class` class to ensure it can be written and read. This is done with a macro:

2b ⟨*Node class info* 2b⟩≡ (18)

```
IRCLASS(Node);
```

The definitions for the `_` field and `className` are standard and tedious to type, so other files may use the `IRCLASSDEFS` macro:

2c ⟨*IRCLASSDEFS macro* 2c⟩≡ (17b)

```
#  define IRCLASSDEFS \
  public: \
    static IR::Class _; \
    virtual const std::string className() const { return _; }
```

1.2 An example user-defined Node

```
class MyNode : public Node {
  IRCLASSDEFS;  // Class name machinery: sets public:
  Node *f1;
  string f2;
  vector<MyNode*> f3;

  protected:
    void read(XMListream &r) {
      Node::read(r);    // Read fields of parent class
      r >> f1 >> f2 >> f3; // Read our fields
    }
    void write(XMLostream &w) const {
      Node::write(w);   // Write parent class
      w << f1 << f2 << f3; // Write our fields
    }
};

/* In a .cpp file */
IRCLASS(MyNode);
```

1.3 The Class class

While writing XML, the system needs to know the name of the class of each object, and during reading, needs a way to create an IR object given only its name. The `Class` class performs this feature.

The class has a static *map* called `classmap` that keeps a pointer to a zero-argument function that creates a new IR node object for each class name. The `newNodeByName` method uses this to create a new `Node` object (usually, something derived from `Node`).

The other part of the class is a static `string` that holds the name of the class. The `Node::className` method implicitly uses the cast-to-const-string operator to retrieve the name of the class.

3 ⟨*Class class* 3⟩≡ (17b)

```
// Information for the classes in the IR: a string naming each,
// and a map giving the constructor for each class
class Class {
  typedef Node *(*createfunc)();
  typedef std::map<std::string, createfunc> createfuncmap;
  static createfuncmap *classmap;
public:
  Class(std::string, createfunc);
  static Node *newNodeByName(const std::string);

private:
    const std::string _className;
  public:
    operator const std::string () { return _className; }
  };
```

As mentioned above, each `Node` has a static `Class` member that holds its name. The constructor for this takes both a string naming the class and the address of a no-argument function (`Class::createfunc`) for constructing it. The `IRCLASS` macro does this automatically using the `constructor` template, which generates such a function automatically.

4a ⟨*IRCLASS macro* 4a⟩≡ (17b)

```
# define IRCLASS(s) IR::Class s::_ = IR::Class( #s, IR::constructor<s> )
```

4b ⟨*constructor template* 4b⟩≡ (17b)

```
template<class T> Node* constructor() { return new T(); }
```

The constructor for `Class` saves the name of the class and enters it into the map along with the constructor function.

A pointer is used for `classmap` to insure the correct order of initialization. The map must be constructed before the `Class` constructor enters anything into it, but C++ does not provide any such guarantees. Instead, the pointer to `classmap` is initialized to 0 (such constant data is typically initialized before the program starts in the data section) and the constructor (somewhat wastefully) checks the pointer each time a new class is registered.

4c ⟨*Class constructor* 4c⟩≡ (18)

```
  Class::createfuncmap *Class::classmap = 0;

  Class::Class(const std::string s, Class::createfunc f) : _className(s)
  {
    if (!classmap) classmap = new createfuncmap();
    (*classmap)[s] = f;
  }
```

The `newNodeByName` method uses the `classmap` map to construct an object whose class name is the given string. For this to work, the class must derived from `Node` and the type registered by a call to the `CLASS` constructor (i.e., by using `IRCLASS`).

4d ⟨*Class::newNodeByName* 4d⟩≡ (18)

```
  Node * Class::newNodeByName(const std::string s)
  {
    assert(classmap);
    createfuncmap::iterator i = classmap->find(s);
    if (i != classmap->end()) {
      createfunc cf = (*i).second;
      return (*cf)();
    } else {
      throw Error("Unknown class " + s);
    }
  }
```

2 Writing XML

Writing XML is fairly easy. The class encapsulates an *ostream* and provides `<<` operators that behave like those for standard streams. So a Node's `write` method can be defined as shown in Section 1.2.

This class is meant to be used as follows:

```
Node *root;

/* Build a (possibly cyclic) graph of Nodes under root ... */

XMLostream w(std::cout);
w & root;
```

Writing a Node object starts a depth-first search of all its fields. To avoid writing duplicate copies of the same object, the `ID` map maintains a positive integer identifier for each written object. When the DFS encounters an object it has already written, a reference to that object's ID is written, rather than the information in the node itself.

5 ⟨*XMLostream class* 5⟩≡ (17b)

```
class XMLostream {
  typedef std::map<const Node *, unsigned int> idmap;
  idmap ID;

  unsigned int nextID;
  std::ostream &o;

public:
  XMLostream(std::ostream &oo) : nextID(0), o(oo) {}
  XMLostream& operator <<(const Node&);
  XMLostream& operator <<(const Node *);
  XMLostream& operator <<(const std::string);
  XMLostream& operator <<(int);
  XMLostream& operator <<(bool);
  ⟨XMLostream vector output 7d⟩
  ⟨XMLostream map output 8⟩
};
```

The following operator, which writes a single `Node`-derived object, is fairly simple: it just identifies the type of the object and writes its contents. These objects are mean to be fields of other objects, and it is assumed that nothing ever points directly to them.

6a ⟨*XMLostream Node output* 6a⟩≡ (18)

```
XMLostream& XMLostream::operator <<(const Node &n)
{
  o << "<" << n.className() << '>' << std::endl;
  n.write(*this);  // usually a recursive call
  o << "</" << n.className() << ">" << std::endl;
  return *this;
}
```

The following operator, which writes a `Node` object referred to by a pointer, does most of the heavy lifting. Null pointers are written as a single XML element called `NULL`. If this `XMLostream` has already written the given object, an XML element called `Ref` is written whose `id` attribute is the integer identifier of the object.

Otherwise, the operator writes the given object by writing an element named by the object's type (obtained by calling `Node::className`). To write the fields of the object, it calls the `write` method on the given `Node`. This virtual method is expected to use the `&` operator to write the objects in its fields to this stream (see Section 1.2).

6b ⟨*XMLostream Node pointer output* 6b⟩≡ (18)

```
XMLostream& XMLostream::operator <<(const Node *n)
{
  if (n == NULL) {
    o << "<NULL/>" << std::endl;
  } else {
    idmap::iterator i = ID.find(n);
    if (i != ID.end()) {
      // Already written this Node: leave a placeholder
      o << "<Ref id=\"" << (*i).second << "\"/>" << std::endl;
    } else {
      o << "<" << n->className() << " id=\"" << nextID << "\">" << std::endl;
      ID[n] = nextID;
      nextID++;
      n->write(*this);  // usually a recursive call
      o << "</" << n->className() << ">" << std::endl;
    }
  }
  return *this;
}
```

Strings are written by simply sending their output to the output stream. Empty strings are written as an `EmptyString` element because the reader otherwise discards empty strings.

7a ⟨*XMLostream string output* 7a⟩≡ (18)

```
XMLostream& XMLostream::operator <<(const std::string s)
{
  o << "<S>";
  for (std::string::const_iterator i = s.begin() ; i != s.end() ; i++ )
    switch (*i) {
    case '&': o << "&amp;"; break;
    case '<': o << "&lt;"; break;
    case '>': o << "&gt;"; break;
    case '\'': o << "&apos;"; break;
    case '\"': o << "&quot;"; break;
    default: o << *i; break;
    };
  o << "</S>\n";
  return *this;
}
```

Integers are written as text in an `Int` element.

7b ⟨*XMLostream int output* 7b⟩≡ (18)

```
XMLostream& XMLostream::operator <<(int i)
{
  o << "<Int>" << i << "</Int>\n";
  return *this;
}
```

A Boolean is written as either a `BoolTrue` or `BoolFalse` element.

7c ⟨*XMLostream bool output* 7c⟩≡ (18)

```
XMLostream& XMLostream::operator <<(bool b)
{
  if (b) o << "<BoolTrue/>";
  else o << "<BoolFalse/>";
  return *this;
}
```

The contents of a vector is written by writing each of the objects it contains followed by a single `EOV` (End-of-vector) XML element.

7d ⟨*XMLostream vector output* 7d⟩≡ (5)

```
  template <class T> XMLostream& operator <<(const std::vector<T*> &v) {
    typename std::vector<T*>::const_iterator i;
    for ( i = v.begin() ; i != v.end() ; i++ ) (*this) << *i;
    o << "<EOV/>" << std::endl;
    return *this;
  }
```

The contents of a map is written out a sequence of keys and values followed by an `EOM` (End-of-map) XML element.

8 ⟨*XMLostream map output* 8⟩≡ (5)

```
template <class K, class V>
 XMLostream& operator <<(const std::map<K, V> &m) {
  typename std::map<K, V>::const_iterator i;
  for ( i = m.begin() ; i != m.end() ; i++ )
    (*this) << (*i).first << (*i).second;
  o << "<EOM/>" << std::endl;
  return *this;
}
```

3 Reading XML

Reading XML is fairly complex. This system uses the *expat* XML parser[1] to first build an internal tree representing the contents of the XML file, then walks this tree, creating Node-derived objects based on the name of each element, then using each Node's `read` method to "fill in" each object's fields.

This two-phase process is awkward, but the (fairly standard) "event-driven" interface of the *expat* parser almost demands it. Basically, you give *expat* a pointer to a function that it calls each time it encounters an element. While the structure of XML guarantees elements are enountered in a top-down order, *expat* insists that you return from this function before it will parse further, so it is difficult to implement a traditional top-down parser.

The public interface to the class consists of a constructor that takes a standard *istream* and overloadings of the `>>` operator that parses an XML element into a variable passed by reference. The class is meant be used as follows:

```
Node *root;

XMListream r(std::cin);
r >> root;
```

9 ⟨*XMListream class* 9⟩≡ (17b)

```
  class XMListream {
    std::stack<XMLNode*> parents;
    XMLNode *lastsibling;
    XMLNode *current;
    XMLNode *root;

    static void startElement(void *, const char *, const char **);
    static void endElement(void *, const char *);
    static void charData(void *, const XML_Char *, int);
    void attachSibling(XMLNode *);

    // Map that tracks Nodes with IDs
    typedef std::map<const std::string, Node *> nodemap;
    nodemap nodeofid;

    Node *getNextNode();

  public:
    XMListream(std::istream &);
    ~XMListream() { delete root; }

    ⟨XMListream Node ptr input 15b⟩
    ⟨XMListream Node input 15a⟩
    XMListream& operator >>(std::string&);
    XMListream& operator >>(int &);
```

---

[1] http://expat.sourceforge.net/

```
  XMListream& operator >>(bool &);
  ⟨XMListream vector input 16b⟩
  ⟨XMListream map input 16c⟩
};
```

3.1 The XMLNode class

The XMLNode class holds the parsed XML tree that the `&` operators later read. Each node has a name, a body containing its character data, which is often ignored, a map holding the node's attributes, and pointers to its first child and next sibling.

10a ⟨*XMLNode class* 10a⟩≡ (17b)

```
struct XMLNode {
  std::string name;
  std::string body;

  typedef std::map<const std::string, const std::string> attrmap;
  attrmap attrs;

  XMLNode *first;
  XMLNode *next;

  XMLNode() : first(0), next(0) {}

  ~XMLNode() { delete first; delete next; }

  void print();
};
```

The `print` method pretty-prints the tree for debugging.

10b ⟨*XMLNode::print* 10b⟩≡ (18)

```
void XMLNode::print() {
  std::cout << '<' << name;
  for (attrmap::iterator j = attrs.begin() ; j != attrs.end() ; j++)
    std::cout << ' ' << (*j).first << "=\"" << (*j).second << '\"';
  std::cout << '>';
  std::cout << body;
  if (first) first->print();
  std::cout << "</" << name << ">" << std::endl;
  if (next) next->print();
}
```

3.2 Parsing XML with expat

The *expat* parser is invoked in the constructor of `XMListream` to construct a tree of `XMLNode` objects. The functions whose names start with `XML_` are all part of *expat*. The constructor creates a parser, tell it to call the static methods `startElement`, `endElement`, and `charData` with a pointer to itself, then repeatedly fills `buffer` with characters from the input stream and parses it.

11 ⟨*XMListream constructor* 11⟩≡ (18)

```
XMListream::XMListream(std::istream &i)
{
  root = lastsibling = NULL;
  XML_Parser p = XML_ParserCreate(NULL);
  if (!p) throw Error("Couldn't create parser");

  XML_SetElementHandler(p, startElement, endElement);
  XML_SetCharacterDataHandler(p, charData);
  XML_SetUserData(p, (void *) this);

  do {
    static const size_t SIZE = 8192;
    char buffer[SIZE];

    i.read(buffer, SIZE);
    if (i.bad()) throw Error("Read error");
    if (!XML_Parse(p, buffer, i.gcount(), i.eof())) {
      std::ostringstream ost;
      ost << "XML parsing error at line " << XML_GetCurrentLineNumber(p)
          << ':' << XML_ErrorString(XML_GetErrorCode(p));
      throw Error(ost.str());
    }
  } while (!i.eof());

  XML_ParserFree(p);
  if (!parents.empty()) throw Error("Non-empty stack.");

  // root->print(); // For debugging
  current = root;
}
```

The static methods `startElement`, `endElement`, and `charData` are called by the *expat* parser when it encounters the start or end of an XML element (e.g., `<MyNode>` and `</MyNode>`) or character data appearing between such markers.

During this process, the three methods maintain the stack of `XMLNode` objects that define the path through the tree to the element currently-being parsed. The top of the `parents` stack is the `XMLNode` whose body is currently being parsed, and the pointer `lastsibling` points to its most-recent-added child, which starts out NULL right after a new element is encountered.

The `attachSibling` method attaches the given node at the bottom of the current tree of `XMLNodes`, defined by the `parents` stack. Just after a new node is created, `lastsibling` is NULL and the method attaches the node as the first child at the top of the stack. After that, it simply sets the `next` pointer of the most-recently-added child.

12a ⟨*XMListream attachSibling* 12a⟩≡ (18)

```
void XMListream::attachSibling(XMLNode *n)
{
  if (root == NULL) root = n;
  if (!lastsibling) {
    if (!parents.empty()) parents.top()->first = n;
  } else
    lastsibling->next = n;
  lastsibling = n;
}
```

The `startElement` method creates a new `XMLNode`, calls `attachSibling` to attach it to the tree, then pushes it onto the `parent` stack and clears the last sibling pointer so `attachSibling` will next attach its first child.

12b ⟨*XMListream startElement* 12b⟩≡ (18)

```
void XMListream::startElement(void *rr, const char *name, const char **attrs)
{
  XMListream *r = static_cast<XMListream*>(rr);
  XMLNode *newNode = new XMLNode();
  newNode->name = name;
  while (*attrs) {
    newNode->attrs.insert( std::make_pair(*attrs,*(attrs+1)) );
    attrs += 2;
  }

  r->attachSibling(newNode);

  r->parents.push(r->lastsibling);
  r->lastsibling = NULL;
}
```

The `endElement` method restores the `lastsibling` pointer that was last pushed onto the `parents` stack by `startElement`.

13a ⟨*XMListream endElement* 13a⟩≡ (18)

```
  void XMListream::endElement(void *rr, const char *)
  {
    XMListream *r = static_cast<XMListream*>(rr);
    // discard the topmost sibling; we'll go back to where we were.
    r->lastsibling = r->parents.top();
    r->parents.pop();
  }
```

The `charData` method takes the non-null-terminated character string passed from *expat* (hence the `len` argument) and appends it to the body of the most-recently-created `XMLNode`, if there is one. Appending is necessary because *Expat* may call this multiple times within the same block of text if it contains entities such as `&lt;`.

13b ⟨*XMListream charData* 13b⟩≡ (18)

```
void XMListream::charData(void *rr, const XML_Char *ss, int len)
{
  XMListream *r = static_cast<XMListream*>(rr);
  std::string s(ss, len);
  if (!(r->parents.empty()))
    r->parents.top()->body += s;
}
```

3.3 Parsing XMLNode trees

Once the `XMListream` constructor runs and builds an `XMLNode` tree from the XML file, the `getNextNode` method steps through the tree in a depth-first order under the control of the IR Node's `read` methods. During this process, the `current` pointer points to the node in the `XMLNode` tree currently being read, and the `parents` stack is used to track the current path from the root.

14 ⟨*XMListream getNextNode* 14⟩≡ (18)

```
Node *XMListream::getNextNode()
{
  if (!current) throw Error("Expecting an element, found nothing");
  Node *n;

  XMLNode::attrmap::iterator idit = current->attrs.find("id");

  if (current->name == "NULL") {       // NULL pointer

    n = NULL;

  } else if (current->name == "Ref") { // Reference to existing object

    if ( idit == current->attrs.end() )
      throw Error("Ref node without id attribute");

    nodemap::iterator ni = nodeofid.find((*idit).second);
    if ( ni == nodeofid.end())
      throw Error("Ref to undefined node id " + (*idit).second);
    n = (*ni).second;

  } else {                            // Normal object

    std::string name = current->name;
    n = Class::newNodeByName(name);
    if (idit != current->attrs.end()) nodeofid[(*idit).second] = n;
    parents.push(current);
    current = current->first;
    n->read(*this);    // Fill in the node's fields from this stream
    if (current != NULL) throw Error("excess elements under " + name);
    current = parents.top();
    parents.pop();
  }
  current = current->next;
  return n;
}
```

15a ⟨*XMListream Node input* 15a⟩≡ (9)

```
template <class N> XMListream& operator >>(N &n) {
  if ( current->name != std::string(N::_) ) // Check name of this class
    throw Error("Unexpected element " + current->name);
  parents.push(current);
  current = current->first;
  n.read(*this); // Fill in the object's fields
  if (current != NULL)
    throw Error("excess elements for " + std::string(N::_));
  current = parents.top();
  parents.pop();
  current = current->next;
  return *this;
}
```

The pointer version of the `&` operator reads a Node using `getNextNode` and uses `dynamic_cast` to cast it to the type of the referenced object, provided we didn't find a NULL pointer.

15b ⟨*XMListream Node ptr input* 15b⟩≡ (9)

```
  template <class T> XMListream& operator >>(T* &f) {
    Node *n = getNextNode();
    f = (n != NULL) ? dynamic_cast<T*>(n) : NULL;
    if (!f && n) throw Error("Unexpected element " + n->className());
    return *this;
  }
```

The string version of the `>>` operator looks for either an `EmptyString` element or an `XMLNode` of type `Text`.

15c ⟨*XMListream string input* 15c⟩≡ (18)

```
XMListream& XMListream::operator >>(std::string &s)
{
  if (!current) throw Error("Expecting text before end of element");
  if (current->name != "S")
      throw Error("Expecting text, found " + current->name);
  s = current->body;
  current = current->next;
  return *this;
}
```

15d ⟨*XMListream int input* 15d⟩≡ (18)

```
XMListream& XMListream::operator >>(int &i)
{
  if (!current) throw Error("Expecting an Int");
  if (current->name == "Int") {
    std::istringstream iss(current->body);
    iss >> i;
  } else
    throw Error("Expecting Int, found " + current->name);
  current = current->next;
  return *this;
}
```

16a ⟨*XMListream bool input* 16a⟩≡ (18)

```
XMListream& XMListream::operator >>(bool &i)
{
  if (!current) throw Error("Expecting BoolTrue or BoolFalse");
  if (current->name == "BoolTrue") i = true;
  else if (current->name == "BoolFalse") i = false;
  else throw Error("Expecting BoolTrue or BoolFalse");

  current = current->next;
  return *this;
}
```

The vector version of the `>>` operator reads objects using the `>>` operator until it encounters an `EOV` element, which it discards.

16b ⟨*XMListream vector input* 16b⟩≡ (9)

```
  template <class T> XMListream& operator >>(std::vector<T> &v) {
    while (current && current->name != "EOV") {
       T value;
       (*this) >> value;
       v.push_back(value);
    }
    if (!current) throw Error("vector ended without EOV");
    current = current->next;  // Skip the EOV Element
    return *this;
  }
```

The map version of the `>>` operator reads using `getNextNode` until it encounters an `EOM` element, which it discards.

16c ⟨*XMListream map input* 16c⟩≡ (9)

```
  template <class K, class V> XMListream& operator >>(std::map<K, V> &m) {
    while (current && current->name != "EOM") {
       K key;
       V value;
       (*this) >> key >> value;
       m.insert( typename std::map<K,V>::value_type(key, value) );
    }
    if (!current) throw Error("map ended without EOM");
    current = current->next;  // Skip the EOM Element
    return *this;
  }
```

4 Miscellany

4.1 The Error class

`Error` is a simple class holding a string thrown as an exception for parsing errors.

17a ⟨*Error class* 17a⟩≡ (17b)

```
    struct Error {
      std::string s;
      Error(std::string ss) : s(ss) {}
    };
```

4.2 IR.hpp and IR.cpp

All these classes are defined in the `IR` namespace.

17b ⟨*IR.hpp* 17b⟩≡

```
  #ifndef _IR_HPP
  #  define _IR_HPP

  #  include <expat.h>
  #  include <string>
  #  include <map>
  #  include <stack>
  #  include <vector>
  #  include <iostream>

  ⟨IRCLASSDEFS macro 2c⟩
  ⟨IRCLASS macro 4a⟩

  namespace IR {
    class Node;

    ⟨Error class 17a⟩
    ⟨Class class 3⟩
    ⟨XMLostream class 5⟩
    ⟨XMLNode class 10a⟩
    ⟨XMListream class 9⟩
    ⟨Node class 2a⟩
    ⟨constructor template 4b⟩
  }
  #endif
```

18 ⟨*IR.cpp* 18⟩≡

```
#include "IR.hpp"

#include <expat.h>
#include <iostream>
#include <sstream>
#include <cassert>

namespace IR {
  ⟨Node class info 2b⟩

  ⟨XMLostream Node pointer output 6b⟩
  ⟨XMLostream Node output 6a⟩
  ⟨XMLostream string output 7a⟩
  ⟨XMLostream int output 7b⟩
  ⟨XMLostream bool output 7c⟩

  ⟨XMLNode::print 10b⟩

  ⟨XMListream constructor 11⟩
  ⟨XMListream string input 15c⟩
  ⟨XMListream int input 15d⟩
  ⟨XMListream bool input 16a⟩
  ⟨XMListream getNextNode 14⟩
  ⟨XMListream attachSibling 12a⟩
  ⟨XMListream startElement 12b⟩
  ⟨XMListream endElement 13a⟩
  ⟨XMListream charData 13b⟩

  ⟨Class constructor 4c⟩
  ⟨Class::newNodeByName 4d⟩
}
```