CMake/Tests/Swig

TP CMake, Boost/C++ et Swig

Eric Ramat
ramat@lil.univ-littoral.fr

5 novembre 2007

Durée : 7.5 heures

---

1 Sujet

1.1 Objectif

L'objectif de ce TP est de s'initier aux outils de compilation multi-plateformes et multi-langages. L'outil choisi sera CMake. Il permet de gérer la phase de compilation mais aussi l'installation, les tests unitaires et le packaging. Afin de découvrir CMake, nous allons développer une API multi-langages de manipulation d'un fichier XML. Une première version C++ basée sur la libxml++ est à développer (des exemples sont proposés en annexe). Pour des raisons d'optimisation, le client veut que l'analyse XML soit réalisée impérativement en mode SAX. Ensuite, nous étudierons Swig pour développer une API Python et une API Java basées sur l'API C++.

Le développement de l'API sera mené à l'aide des tests unitaires. Nous vous proposons donc une série d'histoires accompagnées d'exemples qui serviront de cas de tests.

Le fichier XML a encapsulé par une API est le résultat de l'exécution d'un programme qui produit des données typées. Ces données ont certaines propriétés que l'on va découvrir au fur et à mesure des histoires.

1.2 Première histoire

Ce premier fichier XML mets en évidence la syntaxe demandée pour la représentation des entiers.

```
<outputData>
  <i name="x">10</i>
</ouputData>
```

Le test doit vérifier que la valeur lue est bien l'entier 10 et que le nom associé est "x". La fonction *long getValue(const std : :string& name) const* sera développée. Elle est appellée après le parsing du fichier. Dès qu'une donnée est présente, elle est accompagnée d'un nom sous forme d'une chaîne de caractères.

On vérifiera avec un second test que si on demande la valeur de "y" la fonction getValue lève une exception.

Remarque : Boost propose des fonctions de parsing de string (*boost : :lexical_cast*). N'hésitez pas à les utiliser car elles soulèvent des exceptions.

1.3 Deuxième histoire

Le fichier ci-dessous est incorrect. La valeur n'est pas du type entier. Testez la levée d'une exception.

```
<outputData>
  <i name="x">10.2</i>
</ouputData>
```

1.4 Troisième histoire

Les valeurs réelles sont introduites dans la syntaxe XML et le client change d'avis sur sa fonction de récupération des valeurs. Si la valeur est un entier alors on fera appel à getLongValue. Si la valeur est réelle alors ce sera getDoubleValue.

```
<outputData>
  <d name="x">10.2</d>
  <d name="y">-5.2</d>
</ouputData>
```

Le test doit vérifier que la fonction retourne la bonne valeur pour "x" et "y".

1.5 Quatrième histoire

Le client désire un nouveau type de valeur : les vecteurs. En voici un exemple :

```
<outputData>
  <v name="v1">
    <d name="x">-0.3</d>
    <d name="y">2.1</d>
  </v>
</ouputData>
```

Le client fait remarquer qu'il a donné pour nom à cette nouvelle valeur, vecteur mais indique que chaque composant de ce vecteur de valeurs réelles portent un nom.

Pour les tests, trois nouvelles fonction sont à développer :

- *vector < double > getVectorValue(std : :string & name) const* pour vérifier que le vecteur existe ;
- *bool isDoubleVector(std : :string & vector_name, std : :string & value_name) const* pour vérifier que l'élément ayant pour nom *value_name* du vecteur est un réel ;
- *double getDoubleVectorValue(std : :string & vector_name, std : :string & value_name) const* pour vérifier un élément du vecteur.

1.6 Cinquième histoire

A ce stade, le client se dit qu'il aimerait bien disposer de vecteurs dont toutes les valeurs ne sont pas du même type. Il propose cet exemple :

```
<outputData>
  <v name="v1">
    <d name="x">-0.3</d>
    <i name="y">1</i>
  </v>
  <v name="v2">
    <i name="z">0</i>
    <i name="t">-6</i>
  </v>
</ouputData>
```

Le client est un peu dépassé par son besoin et ne sait pas quelles sont les implications. Il s'en remet au développeur pour lui proposer des fonctions conformes à ses besoins. Il faut donc tester si le premier élément est un double et le second un entier. Le test sur les valeurs de v1 sera aussi proposé : la première valeur est-elle bien -0.3 et la seconde l'entier 1 ? les noms associés sont-ils bons ?

1.7 Sixième histoire

Comme l'on aurait pu s'y attendre, le client désire généraliser et demande à ce que les valeurs des vecteurs puissent être aussi des vecteurs. Voici un exemple :

```
<outputData>
  <v name="m1">
    <v name="c1">
      <d name="x">-0.3</d>
      <i name="y">1</i>
    </v>
    <v name="c2">
      <d name="x">5.4</d>
      <i name="y">-6</i>
    </v>
  </v>
</ouputData>
```

Proposer les tests de type et sur les valeurs.

1.8 Septième histoire

Le client a une dernière demande. Ses ensembles de valeurs sont indexés comme dans l'exemple ci-dessous :

```
<outputData>
  <step index="0">
    <i name="x">10</i>
  </step>
  <step index="1">
    <i name="x">8</i>
  </step>
</ouputData>
```

Le client précise qu'il veut seulement faire appel au parser à chaque nouvel ensemble. Il aimerait tester la séquence suivante :

- appel du parser ;
- si le parser dispose d'un ensemble complet alors appel d'une méthode qui retourne l'index de l'ensemble ; si l'indexage n'existe pas alors la méthode retourne -1 ;
- appel à la méthode qui retourne le type de x ;
- on reboucle sur la première étape tant que le parser nous dit qu'il y a quelque chose à parser.

Tout le fichier xml est parsé lors du premier appel. Les appels suivants consistent à récupérer les ensembles de valeurs de chaque index.

1.9 Huitième histoire

L'API étant terminé, le client aimerait l'utiliser en Python. Proposer un wrapper python avec Swig et les tests unitaires associés.

2 Annexe 1

L'écriture des exceptions se baseront sur la base runtime_error de la STL. Voici la classe de base :

```
class BaseError : public std::runtime_error
{
  public:
    BaseError(const std::string& argv):std::runtime_error(argv)
        { }
};
```

Si vous voulez développer vos propres exceptions, il reste juste à développer des sous-classes.
Pour les exceptions liées au parsing XML, on utilisera le code suivant :

```
class SaxParserError : public xmlpp::exception
{
  public:
    SaxParserError(const Glib::ustring& argv):xmlpp::exception(argv)
        { }

    virtual ~SaxParserError() throw()
        { }

    virtual void Raise() const
        { throw *this; }

    virtual xmlpp::exception* Clone() const
        { return new SaxParserError(*this); }
};
```

La levée d'une exception est alors de la forme :

```
Throw(utils::SaxParserError, (boost::format("Error '%1%'") % str));
```

3 Annexe 2

L'écriture du parser Sax en C++ passe par l'utilisation de la libxml++. Voici quelques exemples de code pour vous aider à écrire le parser.

3.1 Définition du parser Sax

```
#include <libxml++/libxml++.h>

class MySaxParser : public xmlpp::SaxParser
{
public:
  MySaxParser();
  virtual ~MySaxParser();

protected:
  virtual void on_start_document();
  virtual void on_end_document();
  virtual void on_start_element(const Glib::ustring& name,
                                const AttributeList& properties);
  virtual void on_end_element(const Glib::ustring& name);
};
```

3.2 Un exemple d'utilisation du parser

Le fichier example.xml est analysé incrémentalement par bloc de 63 caractères (ça peut être utile dans le cas de l'analyse d'une trame XML envoyée sur le réseau, par exemple).

```
int main(int argc, char* argv[])
{
  Glib::ustring filepath = "example.xml";
  std::ifstream is(filepath.c_str());
  char buffer[64];

  MySaxParser parser;
  do {
    is.read(buffer, 63);
    Glib::ustring input(buffer, is.gcount());

    parser.parse_chunk(input);
  }
  while(is);

  parser.finish_chunk_parsing();

  return 0;
}
```

3.3 Définition des méthodes du parser

Le constructeur et le destructeur sont ici vides. Ils doivent initialiser les structures nécessaires à l'analyse SAX.

```
MySaxParser::MySaxParser()
  : xmlpp::SaxParser()
{
}

MySaxParser::~MySaxParser()
{
}
```

Les méthodes *on_start_document* et *on_end_document* sont invoquées respectivement à l'ouverture et à la fermeture de la balise root.

```
void MySaxParser::on_start_document()
{
  std::cout << "on_start_document()" << std::endl;
}

void MySaxParser::on_end_document()
{
  std::cout << "on_end_document()" << std::endl;
}
```

A chaque ouverture et fermeture de balise, les méthodes *on_start_element* et *on_end_element* sont appelées. Le code ci-dessous montre comment récupérer les artibuts et les valeurs associées à l'attribut de la balise.

```
void MySaxParser::on_start_element(const Glib::ustring& name,
                                   const AttributeList& attributes)
{
```

```
  std::cout << "node name=" << name << std::endl;

  // Print attributes:
  for(xmlpp::SaxParser::AttributeList::const_iterator iter = attributes.begin();
     iter != attributes.end(); ++iter)
  {
    std::cout << "  Attribute " << iter->name << " = "
              << iter->value << std::endl;
  }
}

void MySaxParser::on_end_element(const Glib::ustring& name)
{
  std::cout << "on_end_element()" << std::endl;
}
```

4 Références

- CMake : http://www.cmake.org
- Boost : http://www.boost.org
- STL : http://www.sgi.com/tech/stl/
- XML++ : http://libxmlplusplus.sourceforge.net
- swig : http://www.swig.org
- Python : http://www.python.org