Introduction à Boost

E. RAMAT et G. QUESNEL

Université du Littoral - Côte d'Opale

15 octobre 2007

Plan

Introduction

- Boost est un ensemble de bibliothèques open-source afin d'étendre les fonctionnalités du langage C++ ;
- les librairies sont sous la licence `Boost Sofware License` : peut être utilisé dans des projets ouverts ou non ;
- plusieurs de ces librairies ont été acceptées pour être intégrés au `Technical report` C++ - spécification des nouveaux ajouts à la librairie standard C++ ;

- multithreading ;
- tests unitaires ;
- containers évolués : graphes, tableaux multi-dimensionnels, ... ;
- métaprogrammation ;
- gestion mémoire : les smart pointers, par exemple ;
- encapsulation du système de fichiers ;
- mathématiques : random, integer, rational, ... ;
- ...

Plan

Introduction

STL

La STL (Standard Template Library) du langage C++ est incomplète. Commençons par ces compléments !

Les chaînes de caractères

- conversion de case : to_upper et to_lower ;
- trim : nettoyage des chaînes de caractères ;
- find : recherche d'occurrences, de patterns, ... ;
- remplacement et suppression ;
- split : découpage d'une chaîne de caractères en sous-chaînes ;

Les chaînes de caractères

Un premier exemple

String

```
#include <boost/algorithm/string.hpp>
using namespace std;
using namespace boost;

string str1("␣hello␣world!␣");
to_upper(str1);  // str1 == " HELLO WORLD! "
trim(str1);      // str1 == "HELLO WORLD!"

string str2 = to_lower_copy(
                ireplace_first_copy(
                 str1,"hello","goodbye"));
// str2 == "goodbye world!"
```

- to_upper ou trim : l'algorithme s'applique sur l'objet passé en paramètre ;
- to_lower_copy travaille sur une copie de l'objet qui est ensuite retourné.

Exemple de nettoyage

```
#include<boost/algorithm/string/trim.hpp>

[...]

std::string str1("␣␣␣␣␣hello␣world!␣␣␣␣␣");
std::string str2(trim_left_copy(str1));
// => str2 == "hello world!     "

std::string str3(trim_right_copy(str2));
// => str3 == "     hello world!"

std::trim(str1);
// => str1 == "hello world!"

std::string phone("00423333444");
trim_left_if(phone, is_any_of("0"));
// => phone == "423333444"
```

Chaînes de caractères

Conversion de chaînes

Les conversions de chaînes de caractères en types simples (entiers, réels, booléens, etc.) et inversement sont des opérations prise en charge :

C : `atoi` (int), `atol` (long), `atoll` (long long), et `atof` (double) basé sur `strtol` : uniquement dans un sens, sinon `sprintf`.

C++ : solution des std::stringstream :

```
std::stringstream stream("123123.343");
double a;
stream >> a;
```

Avantage : gère les différences entre les langues,
Désavantage : lourdeur d'écriture.
Solution : `lexical_cast < destination >(value)`

Chaînes de caractères

Conversion de chaînes

Exemple

```
short test(const std::string& str)
{
  using boost::lexical_cast;
  using boost::bad_lexical_cast;

  short  result;

  try {
    result = lexical_cast < short >(str);
  } catch (const bad_lexical_cast& e) {
    std::cerr << e.what();
  }

  return result;
}
```

Chaînes de caractères

Conversion de chaînes

Exemple

```
std::string str("Erreur␣:");
str += boost::lexical_cast < std::string >(123.4321);
```

Peut être couplé à `numeric_cast` pour s'assurer les conversions de numériques :

Exemple

```
try {
  short s = boost::numeric_cast < short >(1234567);
} catch(const boost::negative_overflow& e) {
      std::cout << e.what();
} catch(const boost::positive_overflow& e) {
      std::cout << e.what();
} catch(const boost::bad_numeric_cast& e) {
      std::cout << e.what();
}
```

Chaînes de caractères

tokenizer

- le package Tokenizer fournit un mécanisme simple pour découper une chaîne de caractères ou une séquence de caractères en “tokens” ;
- un “token” est une sous-chaîne ;
- par défaut, les délimiteurs de “tokens” sont les espaces et les caractères de ponctuation ;

String

```
include<boost/tokenizer.hpp>

int main()
{
  string s = "This␣is,␣␣a␣test";
  tokenizer<> tok(s);

  for(tokenizer<>::iterator it=tok.begin(); it!=tok.end();++it)
    cout << *it << "\n";
}
```

Chaînes de caractères

tokenizer

- la classe boost::char_separator<char> permet de spécifier les caractères de délimitation ;

Tokenizer

```
boost::char_separator<char> sep("-;|");
boost::tokenizer<boost::char_separator<char> > tokens(str, sep);
```

- la spécification des délimiteurs peut embarquer des directives : exclure les “tokens” vides, par exemple

Tokenizer

```
boost::char_separator<char> sep("-;|", boost::keep_empty_tokens);
```

Chaînes de caractères

tokenizer

- la détermination des “tokens” peut être réalisé sur la base d’un ensemble d’offsets ;
- ces offsets indiquent le nombre de caractères constituant le token ;

Tokenizer

```
#include<boost/tokenizer.hpp>

int main()
{
  string s = "12252001";
  int offsets[] = {2,2,4};
  offset_separator f(offsets, offsets+3);
  tokenizer<offset_separator> tok(s,f);

  for(tokenizer<offset_separator>::iterator it=tok.begin(); it!=
    cout << *it << "\n";
}
```

Chaînes de caractères

tokenizer

- le constructeur de offset_separator admet comme paramètres le début et la fin de la liste des offsets ;
- un troisième paramètre, `bwrapoffsets`, est mis à true par défaut pour indiquer que si la chaîne est plus longue que la somme des offsets alors le motif est réappliqué.
- par exemple, "1225200101012002" avec (2,2,4) donne 12 25 2001 01 01 2002.
- si `bwrapoffsets` est faux alors le motif n'est pas réappliqué.

Chaînes de caractères

tokenizer

- il existe une troisième technique : “Escaped List Separator” ;
- c’est une généralisation de l’analyse des CSV (Comma Separated Value) ;
- par défaut, le délimiteur de “tokens” est la virgule ;
- les “tokens” sont organisées en ligne et le séparateur de lignes est le retour chariot ;
- le caractère d’échappement est backslach.

Chaînes de caractères

tokenizer

Tokenizer

```
#include<boost/tokenizer.hpp>

int main()
{
  string s = "Field␣1,\"putting␣quotes␣around␣fields,␣allows␣com
  tokenizer<escaped_list_separator<char> > tok(s);

  for(tokenizer<escaped_list_separator<char> >::iterator
      it=tok.begin();
      it!=tok.end();++it)
    cout << *it << "\n";
}
```

Chaînes de caractères

split

- boost propose aussi une fonction pour le découpage des chaînes : split

Tokenizer

```
string str("hello␣abc-*-ABC-*-aBc␣goodbye");
vector < string > Vec;

// Vec == { "hello abc","ABC","aBc goodbye" }
split(Vec, str, is_any_of("-*"));
```

- le troisième paramètre désigne le prédicat (le test qui sera réalisé sur chaque caractère pour déterminer s'il est ou non un délimiteur) ;

Chaînes de caractères

split

- quelques exemples supplémentaires :
  - is_from_range(char,char) : spécification d'un ensemble à l'aide d'un intervalle ;
  - is_space, is_alpha, is_alnum, is_digit, is_punct, . . .
  - la combinaison des prédicats est possible grâce aux opérateurs &&, || et !

Chaînes de caractères

format

- la bibliothèque format fournit une classe pour les arguments d'une chaîne de formatage (utilisant par exemple par printf) ;
- les arguments sont envoyés dans un flux interne et la totalité des types simples sont autorisés ;
- les types définis par l'utilisateur peuvent aussi être utilisable ;
- l'operateur % est au centre du mécanisme.

Chaînes de caractères

format

- la fonction boost::format admet une chaîne de caractères en paramètre ;
- cette dernière contient des patterns du type d'indiquer l'emplacement des éléments ;
- ensuite une cascade d'opérateurs remplacement ;
- le deuxième, ...

Premier exemple

```
cout << boost::format("writing␣%1%,␣␣x=%2%␣:␣%3%-th␣try") % "toto"
// prints "writing toto,  x=40.230 : 50-th try"
```

Chaînes de caractères

format

Plusieurs types de chaînes de formatage sont possibles :

- une simple numérotation avec indication des emplacements ;
- le type printf (

format et printf

```
cout << format("writing␣%s,␣␣x=%s␣:␣%d-th␣step␣\n")
% "toto" % 40.23 % 50;
```

- le type Posix-printf avec position ([N$] [flags] [width] [. precision] type-char);

format et Posix-printf

```
cout << format("(x,y)␣=␣(%1$+5d,%2$+5d)␣\n") % -23 % 35;
```

Les flags et types

-	alignement à gauche
=	alignement centré
+	affichage du signe des nombres (même positif)
#	affichage de la base utilisée
0	remplissage par des zéros

Les flags et types - suite

x ou p	mode hexadécimal
o	mode octal
e ou f ou g	modes pour les réels
d ou u ou i	modes pour les entiers
s	chaînes de caractères
c	un caractère

- il est aussi possible d'insérer automatiquement des tabulations (%nt ou %nTX, n étant la taille de la tabulation et X le caractère de remplissage).

Chaînes de caractères

format

- comme en C++ et les flux, il est possible d'appliquer des manipulateurs aux chaînes de formatage ;
- la plupart des manipulateurs sont aussi disponibles grâce à la notation posix-printf ;
- l'application des manipulateurs passe par la méthode modify_item, pour la modification d'un élément de la chaîne, ou par l'opérateur

modifiy_item

```
format fmt("_%1%_␣%1%␣\n");
fmt.modify_item(1, group(showpos, setw(5)) );

cout << fmt % 101 ;
```

Chaînes de caractères

format

- la fonction group spécifie les manipulateurs à appliquer et la valeur de l'élément sur lequel ils s'appliquent ;

group

```
cout << format("%1% %2% %1%\n") % group(hex, showbase, 40) % 50;
// prints "0x28 50 0x28\n"
```

Plan

Gestion mémoire

Gestion mémoire

Comment gérer au mieux la mémoire : garbage collecteur, optimisation de l'allocation et de la libération de petits objets, ...

Gestion mémoire

Smart pointers - Définition

- smart pointers = “pointeur intelligent” ;
- le but principal est de déléguer la gestion mémoire à un pointeur intelligent ;
- création d’un sorte de garbage collector ; ne plus se préoccuper de la libération mémoire ;
- cette bibliothèque est un ensemble de classes template ;
- la seule possibilité dans la librairie standard, c’est auto_ptr : une encapsulation du pointeur pour éviter de faire appel à delete.

Gestion mémoire

scoped_ptr

- le type auto_ptr surcharge l'opérateur = ce qui autorise le passage du pointeur caché à un autre auto_ptr.

auto_ptr

```
int *i = new int;
auto_ptr<int> x(i);
auto_ptr<int> y;

y = x;

cout << x.get() << endl;
cout << y.get() << endl;
```

- problème : l'auto_ptr x pointe sur NULL après l'opérateur = ;
- la solution : le scoped_ptr.

Gestion mémoire

scoped_ptr

- scoped_ptr est un template et hérite de noncopyable (interdit le constructeur par recopie et l'opérateur =) ;
- les méthodes :
  - reset : destruction de l'objet pointé et pointe sur un nouvel objet ;
  - operator*, operator-> et get : accès à l'objet ou au pointeur ; une assertion contrôle si le pointeur n'est pas nul pour les deux opérateur ;
  - operator bool et operator! : teste la nullité du pointeur ;
  - swap : échange les pointeurs entre 2 pointeurs intelligents.

Gestion mémoire

scoped_ptr

Enveloppe-Lettre

L'utilisation la plus courante est l'implémentation du paradigme de l'enveloppe/lettre : une classe encapsule la manipulation d'une instance allouée dynamiquement et cache alors l'implémentation.

Gestion mémoire

scoped_ptr

scoped_ptr

```
#include <boost/utility.hpp>
#include <boost/scoped_ptr.hpp>

class example : private boost::noncopyable {
public:
  example(): _imp( new implementation ) {}
  ~example() { }
  void do_something() { std::cout << "did something\n"; }
private:
  class implementation;
  boost::scoped_ptr< implementation > _imp;
};
```

Gestion mémoire

scoped_ptr

scoped_ptr

```
class example::implementation {
public:
  ~implementation()
  { std::cout << "destroying␣implementation\n"; }
};

int main() {
  example my_example;
  my_example.do_something();
  return 0;
};
```

Gestion mémoire

scoped_array

- scoped_array est un template stockant un pointeur sur un tableau alloué dynamiquement et hérite de noncopyable ;
- scoped_array diffère sur la méthode de désallocation delete[] au lieu de delete ;
- il existe une surcharge de l'opérateur [] ;
- scoped_array est une alternative à vector de la librairie standard ; scoped_array manipule des tableaux de taille fixe alors que vector se redimensionne automatiquement.

Gestion mémoire

scoped_array

scoped_array

```
#include <boost/scoped_array.hpp>

class A
{
public:
  A(unsigned int size):array(new int(size))
  {
  }

private:
  boost::scoped_array<int> array;
};
```

Gestion mémoire

shared_ptr

- shared_ptr est un template et encapsule un pointeur avec compteur de références ;
- à chaque affectation, construction par copie, ..., le compteur de réferences est incrémenté ;
- l'objet pointé est désalloué lorsque le nombre de références est nul ;
- le pointeur doit référencer un objet alloué dynamiquement (les tableaux ne sont pas pris en compte - utilisé shared_array dans ce cas).

shared_ptr

```
class example
{
public:
  example():_imp(new implementation) {}
  void do_something()
  { std::cout << "use_count() is "
              << _imp.use_count() << std::endl; }
private:
  class implementation;
  boost::shared_ptr< implementation > _imp;
};

class example::implementation
{
public:
  ~implementation()
  { std::cout << "destroying implementation\n"; }
};
```

shared_ptr

```
int main()
{
  example a;
  a.do_something();
  example b(a);
  b.do_something();
  example c;
  c = a;
  c.do_something();
  return 0;
}
```

- l'implémentation cachée dans l'objet a n'est pas dupliquée lors de la création de b ou l'affectation de c ;
- le compteur de références est égal à 3 à la fin de la fonction main ;
- c'est seulement lors de la destruction de a (et donc après la destruction de b et c) que l'implémentation est désallouée.

Gestion mémoire

shared_ptr - constructeurs

- un constructeur sans paramètre : le compteur de références est nul et ne référence rien ;
- un constructeur admettant un pointeur de Y (le type Y étant égal à T ou convertible à T) ; T* est le type du pointeur encapsulé

constructeur

```
template<class Y> explicit shared_ptr(Y * p);
```

Gestion mémoire

shared_ptr - méthodes

- le constructeur par recopie (CopyConstructive) et l'opérateur d'affectation (Assignable) sont possibles ;
- reset() affecte le pointeur à null et met à zéro le compteur de références ; il existe des variantes de reset avec paramètres, dans ce cas, reset est équivant à swap ;
- swap échange le contenu (pointeur et compteur) des deux shared_ptr ;
- deux méthodes d'indirection (* et ->) afin de simplifier l'utilisation du pointeur caché ;
- deux méthodes sur le compteur de références :
  - unique : le compteur est-il égal à un ?
  - use_count : valeur du compteur

Gestion mémoire

weak_ptr

- weak_ptr est un template et encapsule un pointeur qui référence un objet déjà pointé par des shared_ptr ;
- il est construit à partir d'un pointeur partagé ;
- pourquoi faire ?
- si l'objet est détruit par le shared_ptr qui le référençait alors tout accès à l'objet depuis le weak_ptr soulève une expection `boost::bad_weak_ptr` ;
- l'invocation de la méthode reset sur le pointeur partagé produit naturellement le même effet.

Gestion mémoire

weak_ptr

weak_ptr

```
shared_ptr<int> p(new int(5));
weak_ptr<int> q(p);

// un peu plus tard

if(int * r = q.get())
{
  // utilisation de *r
}
```

- la méthode get retourne le pointeur contenu dans le shared_ptr référencé par le weak_ptr ;
- pas de problème si on ne travaille pas en multitâches ;
- en revanche si un autre processus réalise un reset sur p alors il y a un problème d'accès à l'objet via q puisque p ne pointe plus sur l'objet ;
- si le reset est réalisé après l'appel à get, q n'est plus valide mais r est valide et il est seul à possèder l'adresse de l'objet.

Gestion mémoire

weak_ptr

- la solution consiste à définir localement un pointeur partagé en faisant appel à la méthode lock ;
- la fonction lock vérifie si le pointeur est toujours valide ;
- la fonction lock créé ensuite un pointeur partagé et par conséquent, si un reset est fait par la suite sur p alors r reste valide.

weak_ptr

```
shared_ptr<int> p(new int(5));
weak_ptr<int> q(p);

// un peu plus tard

if(shared_ptr<int> r = q.lock())
{
  // utilisation de *r
}
```

Gestion mémoire

intrusive_ptr

- utiliser si l'objet à pointer possède déjà un mécanisme de compteur de références et que l'on désire encapsule le pointeur en utilisant le mécanisme déjà fournit ;
- lors de la création d'un intrusive_ptr, une fonction non qualifié (globale) est invoquée ; cette dernière fait l'appel adéquat au mécanisme de référencement disponible dans l'objet pointé ;
- la destruction de l'intructive_ptr suit la même logique ;
- si l'objet à pointer ne possède pas de mécanisme de compteur de références alors on utilisera les shared_ptr.

Plan

Qu'est ce donc ?

Le **test unitaire** est une technique qui permet de s'assurer du fonctionnement correct d'une partie déterminée d'un logiciel ou d'une portion d'un programme.

- le code du test est **indépendant** du reste du programme.
  - → utilise uniquement les interfaces fonctionnelles.
- il couvre le maximum de code : **couverture de code**.
  - → tout n'est cependant pas testable, il faut parfois émuler : bases de données, fichiers, web, etc.
- nécessite un ***framework*** :
  - → JUnit (java),
  - → CPPunit (C++),
  - → Pyunit (Python), etc.

→ `Boost::Test`

Suite de test

Les tests sont, en général, de simples assertions inclues dans des suites de tests : **test cases** et **test suites**.

test case : de simple tests, `A == B`, `str == "toto"` etc.

```
BOOST_CHECK(0 == 1);
BOOST_CHECK_THROW(openfile("test.dat"), std::exception);
```

test suite : dirige les tests, réaction fasse aux erreurs, où sortir les erreurs, etc.

```
std::ofstream out("output.log");
unit_test_log::instance().set_log_output(out);
```

Test case : les types d'erreurs

Les trois types de cas de tests :

Type	Définition	Compteur	Comportement
WARN	avertissement	non affecté	continue
CHECK	erreur	affecté	continue
REQUIRE	erreur fatale	affecté	annulé

Note

Le **compteur** décris le nombre d'erreurs dans une suite de test.

Boost::test : API

- `BOOST_[type](test)` : s'assure que le test est vrai
- `BOOST_[type]_MESSAGE(str)` : affiche la chaîne de caractères str
- `BOOST_[type]_EQUAL(x, y)` : s'assure que x égal y
- `BOOST_[type]_CLOSE(x, y, z)` : test d'égalité entre des réels x et y avec une tolérance z
- `BOOST_[type]_THROW(fct, expt)` : s'assure que l'appel à la fonction fct lève une exception expt
- `BOOST_[type]_NO_THROW(fct, expt)` : s'assure que l'appel à la fonction fct ne lève pas une exception expt
- `BOOST_CHECKPOINT(str)` : capture la levée de signaux, division par zéro, boucle infinie, etc.

Remarque

À noter l'existence de la macro `BOOST_MESSAGE(str)` qui montre une chaîne de caractères.

Description de l'exemple

- Dans la méthode *Extrem Programming*, il est recommandé d'écrire les tests avant le code afin de forcer à définir des interfaces de programmation propres.
- Dans l'exemple suivant nous considérons les tests unitaires de type **fonctions** :
  - test.cc contient une suite de tests qui exécute un ensemble de fonction représentant les tests.
  - moyenne.cc la fichier contenant la fonction à tester.

Remarque

Les gestionnaires de projets tel que les **autotools**, **cmake** ou encore **scons** gèrent les *makefile* de tests. Pour gérer l'exemple suivant, nous écrivons un *makefile* « maison ».

Écriture d'un *makefile*

Makefile

```
CFLAGS=-Wall -g -O2
LDFLAGS=
LDFLAGSTEST=-lboost_unit_test_framework

# construction de l'executable
all: exe
exe: moyenne.o
        g++ -o exe moyenne.o $(LDFLAGS)
moyenne.o: moyenne.cc
        g++ -c moyenne.cc $(CFLAGS)

# construction et exécution du test
check: test1
        ./test1
test1: test1.o moyenne.o
        g++ -o test1 test1.o moyenne.o $(LDFLAGSTEST)
test1.o: test1.cc
        g++ -c test1.cc $(CFLAGS)
```

Exemple 1

Fichier de test : test.cc

```
#include <boost/test/unit_test.hpp>

using namespace boost::unit_test_framework;

/* la fonction de test */
void test_moyenne()
{
  std::vector < float > tab;
  BOOST_REQUIRE(moyenne(tab) == 0.0);
  tab += 1.0, 2.0, 3.0;
  BOOST_REQUIRE(moyenne(tab) == 2.0);
}

/* encapsulation du main par boost::unit */
boost::unit_test_framework::test_suite*
init_unit_test_suite(int, char* [])
{
    boost::unit_test_framework::test_suite* test;
    test = BOOST_TEST_SUITE("test");
    test->add(BOOST_TEST_CASE(&test_moyenne));
    return test;
}
```

Exemple 2

Fichier de calcul

```
float moyenne(const std::vector < float >& tab)
{
  float sum = 0.0;
  for (std::vector < float >::const_iterator it = tab.begin();
       it != tab.end(); ++it) {
       sum += *it;
  }

  return sum / tab.size();
}
```

Exemple 2

Fichier de calcul

```
float moyenne(const std::vector < float >& tab)
{
  float sum = 0.0;
  for (std::vector < float >::const_iterator it = tab.begin();
       it != tab.end(); ++it) {
       sum += *it;
  }

  return sum / tab.size();
}
```

Problème

À l'exécution des tests, le premier test plante sur une division par zéro !

Exemple 3

Modification du fichier de calcul

```
float moyenne(const std::vector < float >& tab)
{
  if (not tab.empty()) {
    float sum = 0.0;
    for (std::vector < float >::const_iterator it = tab.begin();
         it != tab.end(); ++it) {
         sum += *it;
    }
    return sum / tab.size();
  } else {
    return 0.0;
  }
}
```

Une autre façon de faire les tests

Utilisation des auto-tests

```
#define BOOST_TEST_MAIN
#define BOOST_AUTO_TEST_MAIN
#define BOOST_TEST_MODULE moyenne_test
#include <boost/test/unit_test.hpp>
#include <boost/test/auto_unit_test.hpp>

BOOST_AUTO_TEST_CASE(moyenne)
{
  std::vector < float > tab;
  BOOST_REQUIRE_EQUAL(moyenne(tab), 0.0);
  tab.push_back(1.0); tab.push_back(2.0); tab.push_back(3.0);
  BOOST_REQUIRE_EQUAL(moyenne(tab), 2.0);
}
```

Une autre façon de faire les tests

Utilisation des auto-tests

```
#define BOOST_TEST_MAIN
#define BOOST_AUTO_TEST_MAIN
#define BOOST_TEST_MODULE moyenne_test
#include <boost/test/unit_test.hpp>
#include <boost/test/auto_unit_test.hpp>

BOOST_AUTO_TEST_CASE(moyenne)
{
  std::vector < float > tab;
  BOOST_REQUIRE_EQUAL(moyenne(tab), 0.0);
  tab.push_back(1.0); tab.push_back(2.0); tab.push_back(3.0);
  BOOST_REQUIRE_EQUAL(moyenne(tab), 2.0);
}
```

Auto-tests

Cette technique permet de réduire le code des tests. La création de la suite de tests est automatique.

Quelques références utiles

- Boost Test Libray

  `http://www.boost.org/libs/test/doc`
- Unit test framework

  `http://www.boost.org/libs/test/doc/components/utf`
- Outils de tests

  `http://www.boost.org/libs/test/doc/components/test_tools`
- La concurrence

  `http://cppunit.sourceforge.net/`

Plan

Interopérabilité Python - C++

Introduction

- python possède un mécanisme de couplage avec C/C++ ;
- la bibliothèque boost::python est un “framework” d’interfaçage entre Python et C++ ;
- elle permet d’utiliser des classes et fonctions C++ dans Python et vice-versa sans utiliser d’outil spécifique ;
- la technique repose sur une encapsulation des interfaces C++ sans modification du code (non intrusif) ;
- cette bibliothèque utilise des techniques avancées de métaprogrammation qui permet de simplifier l’écriture ;
- une sorte de langage déclaratif (de type IDL) est utilisé.

Interopérabilité Python - C++

Un exemple : Hello World !

La fonction C

```
char const* greet()
{
  return "hello, world";
}
```

Le wrapper

```
#include <boost/python.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE(hello)
{
  def("greet", greet);
}
```

Une librairie dynamique doit être produite.

Compilation

gcc -shared -o hello.so hello.cpp -I/usr/include/python2.4 -lboost_python-gcc-1_33_1

Le wrapper

```
>>> import hello
>>> print hello.greet()
hello, world
```

La classe C++

```
class World {
  private:
    std::string msg;

  public:
    World() { }
    World(std::string msg):msg(msg) { }
    void set(std::string msg) { this->msg = msg; }
    std::string greet() { return msg; }
};
```

- les paramètres des méthodes et constructeurs ne peuvent pas être passés par référence.
- le constructeur par défaut est obligatoire ! Attention, si vous définissez un autre constructeur, le constructeur par défaut n'existe plus !

L'interface

```
#include <boost/python.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE(hello)
{
  class_<World>("World")
    .def("greet", &World::greet)
    .def("set", &World::set)
  ;
}
```

Constructeur

Le constructeur par défaut (sans paramètre) est défini sans aucune spécification dans l'interface. On peut donc écrire : a = hello.world().

Interopérabilité Python - C++

Les classes - les constructeurs

Une classe C++

```
class World
{
  World(std::string msg): msg(msg) {}
  ...
};
```

L'interface

```
BOOST_PYTHON_MODULE(hello)
{
  class_<World>("World", init<std::string>())
  ...
}
```

Interopérabilité Python - C++

Les classes - les constructeurs

- en Python, le constructeur a pour nom __init__ ;
- une autre possibilité existe par un .def() ;
- .def(init<double, double>()) définit un constructeur à deux paramètres double ;
- pour le constructeur par défaut, on écrira : .def(init<>()) ;

Interopérabilité Python - C++

Les classes - les attributs

- par défaut, les attributs C++ ne sont pas accessibles en Python ;
- deux directives permettent de déclarer l'accès aux attributs :
  - .def_readonly : accès en lecture seule ;
  - .def_readwrite : accès en lecture-écriture ;

Attributs

```
class Var
{
  Var(std::string name) : name(name), value() {}

  std::string const name;
  float value;
};
```

Interopérabilité Python - C++

Les classes - les attributs

L'interface

```
BOOST_PYTHON_MODULE(var)
{
  class_<Var>("Var", init<std::string>())
    .def_readonly("name", &Var::name)
    .def_readwrite("value", &Var::value)
  ;
}
```

Interopérabilité Python - C++

L'héritage

L'interface

```
class A { ... };
class B:public A { ... };

BOOST_PYTHON_MODULE(essai)
{
  class_<A>("A")
    ...
  ;
  class_<B, bases<A> >("B")
    ...
  ;
}
```

Constructeur

Toutes les méthodes héritées sont encapsulées de manière automatique. Si une fonction f est définie dans la classe A alors la classe B (python) disposera de f sans rien spécifier.

Les méthodes virtuelles

```
class A {
public:
  virtual int f() = 0;
};
```

- on pourrait appliquer la même technique que précédemment ... en laissant boost::python encapsuler toutes les méthodes hérités (même abstraite) ;
- dans ce cas, c'est python qui gère le polymorphisme ; on respecte alors la non instrusion de code mais on n'utilise pas le polymorphisme C++ ;

Les méthodes virtuelles

```
class AWrapper : public A, public wrapper<A> {
public:
  int f() {
    return this->get_override("f")();
  }
};

BOOST_PYTHON_MODULE(essai)
{
  class_<AWrapper, boost::noncopyable>("A")
    .def("f", pure_virtual(&Base::f))
  ;
}
```

- la fonction *get_override* a pour objectif de faire "l'aiguillage" ;
- *pure_virtual* indique que la méthode f est abstraite et que A n'est pas instantiable ce qui n'est pas le cas dans la méthode classique.

Les méthodes virtuelles

```
class AWrapper : A, wrapper<A>
{
  int f()
  {
     if (override func = this->get_override("f"))
         return func();
     return A::f();
  }

  int default_f() { return this->A::f(); }
};

BOOST_PYTHON_MODULE(essai)
{
  class_<AWrapper, boost::noncopyable>("A")
   .def("f", &A::f, &AWrapper::default_f)
  ;
}
```

- si la méthode n'est pas abstraite et si elle est surchargée, le wrapper doit tester qui possède la dernière implémentation ;
- l'appel à la fonction *get_override* permet de savoir si la méthode a été surchargée en retournant le pointeur sur cette dernière ;
- la méthode *default_f* encapsule l'appel par défaut de la méthode f définie dans la classe de base A.