7. CTest et Boost

7.1 Documentation

Etudiez le support CMake.pdf distribué précédemment, la documentation dans le fichier /usr/share/doc/cmake/ctest.html, ainsi que les Cours et TP de M1 mis à votre disposition pour « rappel ».

N'oubliez pas d'installer « boost » via la commande suivante :
aptitude install libboost-dev libboost-test-dev libboost-doc

Pour une installation « complête », vous pouvez utiliser la commande suivante :
aptitude install libboost-all-dev

8. Utilisation de Swig

SWIG (Simplified Wrapper and Interface Generator) permet d'interfacer des langages interprétés tels que Perl, Python ou Ruby avec C/C++. Swig utilise les fichiers « header » de ces langages pour générer un emballage (wrapper) qui permet d'accéder aux méthodes, structures et objets des langages C/C++. Swig permet de profiter de l'efficacité des langages compilés C/C++ et de la diversité des différentes bibliothèques construites dans ces langages.

Nous utilisons cet outil pour wrapper (embarquer) du code C/C++ pour python...

Installez swig via la commande suivante :
aptitude install swig swig-doc swig-examples

Consultez la documentation et les exemples dans les sous-répertoires /usr/share/doc/swig*.

Note sur la gestion de la mémoire :

Lorsqu'il n'y a plus de référence sur un objet, le Garbage Collector de Python le supprime. Le destructeur de cet objet Python commande à la bibliothèque C/C++ la libération de la mémoire via le destructeur programmé en C/C++.

8.1 Petit exemple 1 – Code en langage « C »

Voir le sous-répertoire exemple1. Il contient un Makefile (à étudier !!!) ; tapez « make help » pour les détails. La commande « make » sans argument lance le wrapping et la compilation. A l'issue de la compilation nous obtenons :

- Un fichier « hello.py » qui fait office d'interface Python ;
- Un fichier « _hello.so », chargé par « hello.py », contenant nos variables et fonctions en « C ». ATTENTION : Le nom de la librairie doit être préfixée par underscore (« _ »).

Juste un test en passant... Essayez les commandes suivantes :

- file *.o _hello.so
- nm hello.o
- nm _hello.so

Lançons Python (pas trop loin quand même...) :

```
ma_machine$ python
Python 2.5.4 (r254:67916, Sep 26 2009, 08:19:36)
[GCC 4.3.4] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from hello import *
>>> dd = personne()
>>> dd.nom="DUVIVIER"
>>> dd.prenom="D"
>>> hello(dd)

     Hello, D DUVIVIER !

9
>>> hello (cvar.tux)

     Hello, Tux Le Manchot !

13
>>> quit()
ma_machine$
```

Remarque : vous pouvez également utiliser la commande « make run » ou « python testhello.py », étudiez le contenu des divers fichiers (*.h, *.c, *.i, *.py).

8.2 Exercice 1

Le fichier Makefile fourni est désastreux !!! Améliorez-le en vous basant sur les TP précédents et en vous inspirant du modèle suivant (attention, non nécessairement complet et/ou exempt d'erreur !):

```
CC=GCC
CFLAGS=-I/usr/include/python2.5

all: _hello.so

_hello.so: hello.o hello_wrap.o
     $(CC) -shared hello.o hello_wrap.o -o _hello.so

hello_wrap.c: hello.i hello.h
     swig -python -shadow hello.i

clean: ...
```

Vous pouvez le transformer en fichier CMake (CmakeLists.txt).

8.3 Petit exemple 2 – Code en langage « C++ »

Voir le sous-répertoire exemple2. Il contient un Makefile (à étudier !!!), tapez « make help » pour les détails. La commande « make » sans argument lance le wrapping et la compilation. A l'issue de la compilation, nous obtenons les mêmes fichiers que pour l'exemple 1 (hormis hello_wrap qui porte l'extension « .cpp » et non « .c » (...) bien sûr).

Lançons Python (il faudra aller le rechercher, après...) :

```
ma_machine$ python
Python 2.5.4 (r254:67916, Sep 26 2009, 08:19:36)
[GCC 4.3.4] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import hello
>>> dd=hello.personne("D", "DUVIVIER")
>>> dd.hello()
Hello D DUVIVIER
9
>>> hello.cvar.tux.hello()
Hello Tux Le Manchot
13
>>> quit()
ma_machine$
```

Remarque : vous pouvez également utiliser la commande « make run » ou « python testhello.py », étudiez le contenu des divers fichiers (*.hpp, *.cpp, *.i, *.py).

Tout semble aller pour le mieux dans le meilleur des mondes... Dans le fichier d'interface Python (hello.py), une classe Python est déclarée et chacune de ses méthodes appelle la méthode C++ correspondante :

```
class personne(_object):
    ...
    def __init__(self, *args):
        this = _hello.new_personne(*args)
        try: self.this.append(this)
        except: self.this = this
    def prenom(*args): return _hello.personne_prenom(*args)
    def nom(*args): return _hello.personne_nom(*args)
    def hello(*args): return _hello.personne_hello(*args)
    def nommer(*args): return _hello.personne_nommer(*args)
```

Allons un peu plus loin en Python en tentant de renommer un objet « personne » :

```
ma_machine$ python
Python 2.5.4 (r254:67916, Sep 26 2009, 08:19:36)
[GCC 4.3.4] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import hello
>>> dd=hello.personne("D", "DUVIVIER")
>>> dd.hello()
Hello D DUVIVIER
9
>>> dd.nommer("D.", "D.")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "hello.py", line 64, in nommer
    def nommer(*args): return _hello.personne_nommer(*args)
TypeError: in method 'personne_nommer', argument 2 of type 'std::string const'
```

Aie ! Même si nous demandons le nom de tux ça plante :

```
>>> hello.cvar.tux.nom()
<Swig Object of type 'std::string *' at 0xb7f0da48>
>>> quit()
ma_machine$
```

Pourquoi ? Swig sait convertir sans problème les types de base du langage « C », dont les « char * », mais il ne sait pas qu'une chaîne de caractères Python constitue une base valable pour construire une std::string, ni comment convertir l'un en l'autre !

Pour cela, Swig met à notre disposition les « typemaps » qui permettent de traiter les conversions de types au plus bas niveau. ATTENTION aux erreurs (fuites mémoire & cie à ce niveau). Un typemap est une déclaration placée dans le fichier « hello.i » permettant de gérer les passages d'arguments :

```
%typemap(langage, methode) type
{
      /* Code C/C++ */
}
```

où :

- langage : langage cible
- methode : précise pour quelle méthode de conversion ce typemap est écrit (in, out...), cherchez un peu dans les documentations
- type : indique à quel type de données le typemap s'applique

Il faut donc écrire (dans hello.i) au moins un typemap du genre :

```
%typemap(python, in) std::string
{
  --> Ajoutez le code pour convertir si possible
  --> ou afficher un message d'erreur (explicite) sinon
}
```

...