Boost.Python library (C++ / Python)

Purpose of this document

Boost.Python is a C++ library which enables interoperability between **C++** and **Python**. Boost.Python is a part of **Boost libraries** which provides free portable C++ source libraries. The goal of this document is to explain through a simple example how to install and use Boost.Python to connect a **C++** library with a **Python** script.
We will work with the C++ library **AddNumbers**: **AddNumbers.tar.bz2**. Read the **Create and use static and shared C++ libraries** tutorial that explains how to create, compile and link a shared library.
You can download the **Boost.Python_AddNumbersClient** example: **Boost.Python_AddNumbersClient.tar.bz2**.

OPERATING SYSTEM	VERSION
Linux RH Enterprise 4	kernel 2.6.9-11
PROGRAM	VERSION
g++	3.4.3
python	2.3.4
Boost.Jam	3.1.13-1
Boost.Python	1.33.1

Install Boost libraries

These paragraph is intended to help you to install and compile **Boost libraries**. Refer to the **getting started** page for more details. The recommended way to build and install the Boost libraries is to use the **Boost.Build** system. First download and install a **prebuilt executable of Boost.Jam** which is an extension of the **Perforce Jam** portable **make** replacement.

```
[~/softs/boost-jam] > tar -zxvf boost-jam-3.1.13-1-linuxx86.tgz
...
[~/softs/boost-jam] > rm -f boost-jam-3.1.13-1-linuxx86.tgz
[~/softs/boost-jam] > mv boost-jam-3.1.13-1-linuxx86 3.1.13-1
[~/softs/boost-jam] > ln -s 3.1.13-1 default
```

Now download the **Boost libraries**.

```
[~/softs/boost] > tar -jxvf boost_1_33_1.tar.bz2
...
[~/softs/boost] > rm -f boost_1_33_1.tar.bz2
[~/softs/boost] > mv boost_1_33_1.tar.bz2 1.33.1
[~/softs/boost] > ln -s boost_1_33_1.tar.bz2 default
```

Then build and install it using the **bjam** executable.

```
[~/softs/boost/1.33.1] > bjam --prefix=${HOME}/softs/boost/1.33.1 --with-python-version=2.3 install
```

Don't forget to define the **BOOST_ROOT** environment variable that contains the path to the boost libraries.

```
[~] > export BOOST_ROOT=${HOME}/softs/boost/default
```

Create a wrapper

First let us look furtively the C++ class interface we want to work with.

~/workspace/c++/AddNumbers/inc/AddNumbers.h

```
#ifndef _ADDNUMBERS_H
#define _ADDNUMBERS_H

class AddNumbers
{
        private:
        int _a;
        int _b;

        public:
        AddNumbers ();
        ~AddNumbers ();
        void setA (int a);
        void setB (int b);

        int getA () const;
        int getB () const;

        int getSum () const;

}; // AddNumbers

#endif // _ADDNUMBERS_H
```

Now we write a wrapper. See the **tutorial introduction** for more details. In our example we want to expose some methods to the wrapper.

~/workspace/python/Boost.Python_AddNumbersClient/src/AddNumbers_wrap.cpp

```
#include "AddNumbers.h"

#include <boost/python.hpp>

using namespace boost::python;

BOOST_PYTHON_MODULE(AddNumbers_wrap)
{
    class_<AddNumbers>("AddNumbers")
      .def("setA", &AddNumbers::setA)
      .def("setB", &AddNumbers::setB)
      .def("getA", &AddNumbers::getA)
      .def("getB", &AddNumbers::getB)
      .def("getSum", &AddNumbers::getSum)
    ;
}
```

Note that we didn't need to expose the **default constructor** and the **destructor** of the class. Boost.Python do this work by default. Another important remark is that it is recommended to not use the same name for the Python module name (**AddNumbers_wrap**) than the class name (**AddNumbers**). Now compile the wrapper.

```
[~/workspace/python/Boost.Python_AddNumbersClient] > g++ -I $BOOST_ROOT -I /usr/include/python2.3 \
> -I ../../c++/AddNumbers/inc -fpic -c src/AddNumbers_wrap.cpp -o obj/AddNumbers_wrap.o
```

And finally link it as a shared library.

```
[~/workspace/python/Boost.Python_AddNumbersClient] > g++ -shared -L ${BOOST_ROOT}/lib -lboost_python-gcc \
> -L ../../c++/AddNumbers/lib -lAddNumbers -o lib/AddNumbers_wrap.so obj/AddNumbers_wrap.o
```

Don't forget to add the path of the **libAddNumbers.so** library to the **LD_LIBRARY_PATH** environment variable.

```
[~] > export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${HOME}/workspace/c++/AddNumbers/lib
```

As an example see the **Makefile** file in **Boost.Python_AddNumbersClient.tar.bz2**.

Connect a Python script

Let's write the following Python script.

~/workspace/python/Boost.Python_AddNumbersClient/src/AddNumbersClient.py

```
#!/usr/bin/python

import sys
sys.path.append('lib')

import AddNumbers_wrap

ab = AddNumbers_wrap.AddNumbers()
ab.setA(4)
ab.setB(3)
print '%d + %d = %d' % (ab.getA(), ab.getB(), ab.getSum())
```

Then launch it.

```
[~/workspace/python/Boost.Python_AddNumbersClient] > src/AddNumbersClient.py
4 + 3 = 7
```

Install the Pyste code generator

Pyste is a **Boost.Python** code generator. The user specifies the classes and functions to be exported using a simple interface. Then Pyste parse all the headers and extract the information needed to automatically generate C++ code. To run Pyste we need:

- at least Python **2.2**,
- the **elementtree** library,
- the **GCC-XML** parser.

First build and install the Pyste module from the Boost directory.

```
[~/softs/boost/default/libs/python/pyste/install] > python setup.py build install --prefix=$HOME
...
```

The module is installed in the **Pyste** subdirectory of the **${HOME}/lib/python2.3/site-packages** directory. It must be accessible through the **PYTHONPATH** environment variable.

```
[~] > export PYTHONPATH=${HOME}/lib/python2.3/site-packages
```

Then dowload the **elementtree** library.

```
[~/softs/elementtree] > tar -zxvf elementtree-1.2.6-20050316.tar.gz
...
[~/softs/elementtree] > rm -f elementtree-1.2.6-20050316.tar.gz
[~/softs/elementtree] > mv elementtree-1.2.6-20050316 1.2.6-src
```

Now build and install it.

```
[~/softs/elementtree/1.2.6-src] > python setup.py build install --prefix=$HOME
...
```

The library is installed in the **${HOME}/lib/python2.3/site-packages/elementtree** directory. Its parent directory is already accessible in the **PYTHONPATH** environment variable. Then download the **GCC-XML** parser.

```
[~/softs/gccxml] > tar -zxvf gccxml-0.6.0-x86-linux.tar.gz
...
[~/softs/gccxml] > rm -f gccxml-0.6.0-x86-linux.tar.gz
[~/softs/gccxml] > mkdir 0.6.0
[~/softs/gccxml] > ln -s 0.6.0 default
[~/softs/gccxml] > mv README gccxml-0.6.0-x86-linux-files.tar 0.6.0
[~/softs/gccxml] > cd 0.6.0
[~/softs/gccxml/0.6.0] > tar -xvf gccxml-0.6.0-x86-linux-files.tar
...
[~/softs/gccxml/0.6.0] > rm -f gccxml-0.6.0-x86-linux-files.tar
```

Don't forget to make **gccxml** accessible in the **PATH** environment variable.

Using Pyste

Pyste inputs interface files to generate C++ code. See the **Pyste documentation** for more details about creating those interface files. Let us create a such interface file from **AddNumbers.h**.

~/workspace/python/Boost.Python_AddNumbersClient/int/AddNumbers.pyste

```
Class("AddNumbers", "AddNumbers.h")
```

We need only to declare the class **AddNumbers** that we want to expose to the wrapper. Now we generate the C++ Boost.Python code using the **pyste.py** script.

```
[~/workspace/python/Boost.Python_AddNumbersClient] > python ${HOME}/lib/python2.3/site-packages/Pyste/pyste.py \
> --gccxml-path=${HOME}/softs/gccxml/default/bin/gccxml \
> -I ../../c++/AddNumbers/inc --module=AddNumbers_wrap --out=src/AddNumbers_wrap_pyste.cpp int/AddNumbers.pyste
```

Let us compile the generated C++ wrapper **AddNumbers_wrap_pyste.cpp**.

```
[~/workspace/python/Boost.Python_AddNumbersClient] > g++ -I $BOOST_ROOT -I /usr/include/python2.3 \
> -I ../../c++/AddNumbers/inc -fpic -c src/AddNumbers_wrap_pyste.cpp -o obj/AddNumbers_wrap_pyste.o
```

Then we link the shared library.

```
[~/workspace/python/Boost.Python_AddNumbersClient] > g++ -shared -L ${BOOST_ROOT}/lib -lboost_python-gcc \
> -L ../../c++/AddNumbers/lib -lAddNumbers -o lib/AddNumbers_wrap.so obj/AddNumbers_wrap_pyste.o
```

And we execute the python script connected to the wrapper library.

```
[~/workspace/python/Boost.Python_AddNumbersClient] > src/AddNumbersClient.py
4 + 3 = 7
```

As an example see the **Makefile_pyste** file in **Boost.Python_AddNumbersClient.tar.bz2**. Don't forget to define **PYSTEPATH** and **GCCXMLPATH** environment variables before to use the **make** program.

Useful options of bjam

```
bjam [option] ... [install|stage]
```

Boost.Jam is an extension of **Perforce Jam** portable **make** replacement. It contains significant improvements made to facilitate its use in the **Boost.Build** system, but should be backward compatible with Perforce Jam.

OPTION	DESCRIPTION
--prefix=**PREFIX**	Install architecture independent files here. Default on **Windows** platform: **PREFIX**=C:\Boost. Default on **Unix**, **Linux**: **PREFIX**=/usr/local.
--with-python-version=**version**	Build Boost.Python libraries with the Python version indicated. Default is **version**=2.4.
ACTION	**DESCRIPTION**
install	Builds and installs Boost libraries and headers.
stage	Builds the Boost libraries and copies them into a common directory.

Useful options of pyste

```
pyste [option] ...  <interface-file> ...
```

Pyste is a **Boost.Python** code generator. It uses the **gccxml** program to parse all the headers defined in the **interface-file** files and extract the information needed to automatically generate C++ code.

OPTION	DESCRIPTION
--module=<**module-name**>	The name of the module that will be generated; defaults to the first interface filename, without the extension.
-I <**include-path**>	Add an include path.
--out=<**output-file**>	Specify output filename (default: <**module-name**>.cpp).
--gccxml-path=<**gccxml-path**>	Path to **gccxml** executable (default: **gccxml**).

Useful links

LINK	COMMENT
GNU GCC manual	**g++** manual
Boost.Python library	web site of the Boost.Python library
Pyste	Boost.Python code generator
elementtree	**elementtree** library (needed by Pyste)
GCC-XML	**GCC-XML** parser (needed by Pyste)
Boost libraries	web site of the Boost librairies

Last modified on Thursday February 22th, 2007 any comment to **roche@iram.fr**