How to create a project with ParadisEO
How to contribute to ParadisEO

1 Introduction

This package aims at helping you to create a project or a contribution with a minimal test suite (for reliance) which can easily reusable by other users of **ParadisEO**.

2 Content

2 directories :

- MyProject : with files to fill
- Example : an example which can help you to fill "MyProject"

3 What do you have to do

If you encounter some problems to understand something in the following of this document, many tutorials are available on **ParadisEO** website and can help you. You can also contact the **ParadisEO** team.

Il you have a doubt to fill something, see how it has been done in the directory Example.

Your work can be separated in five tasks :

- Configuration
- Build a library with your own sources
- Documentation
- Tests
- Example

4 Configuration

First, you have to link ParadisEO to your projet. Basicaly, you only have to specified components your would like to add to you projet (all components are added by default). In *MyProject/CMakeLists.txt* :

```
find_package(Paradiseo COMPONENTS eo eoutils moeo)
```

If you don't have installed ParadisEO in standard directories, you have to define the root of your installation, for instance :

```
set(PARADISEO_ROOT /home/quemy/paradiseo/)
```

For further information about how to link **ParadisEO**, please, refer to the tutorial on **ParadisEO**

website.

Then, specify the project name, the package name and the version number (project and package name can be the same) in the same file :

```
# Here define your project name
project(projectName)

# Here define the name and the version of your package
set(PACKAGE_NAME "packageName" CACHE STRING "Package name" FORCE)
set(PACKAGE_VERSION "versionNumber" CACHE STRING "Package version" FORCE)
```

5 Build a library with your own sources

For this part, look at *Example/src/* directory to have an example.

For more readability, split your code in maximum of files. Try to have a ".h" or ".hpp" file and a ".cpp" file for each classes (it can be impossible if you use templates, except with the new C++ standard).

Then put all your source code in the MyProject/src directory.

Now you have to complete the *CMakeLists.txt* file :

- Include source directories you need

```
include_directories(${PARADISEO_INCLUDE_DIR}) # include ParadisEO
include_directories({CMAKE_SOURCE_DIR}/src) # include your own source
```

- Set where the library must be compiled

```
set(LIBRARY_OUTPUT_PATH ${CMAKE_BINARY_DIR}/lib}
```

- Add all your ".cpp" files in a list (you can choose the list name)

```
set(LIST_NAME
file1.cpp
file2.cpp
...
fileN.cpp)
```

- Declare what is in your library : all your ".cpp" files define in a list (you can choose the library name)

```
add_library(lib_name STATIC ${LIST_NAME})
```

Now you can normally build your library (build the target all).

6 Documentation

Documentation is very important when you make available your work. It allows users to quickly know how to use your project.

So each file has to be documented :
- Do a short description of each class
- For each function specify :
  - What it does
  - Parameters (@param...)
  - The return value (@return)

    See in the Example sources files of in doxygen documentation

- In the file *MyProject/doc/index.h*, fill the sections "Introduction" and "Authors"

Documentation can be built now (build the target doc)

7 Tests

There are many concepts of tests in computer science. Here we just attempted to do some tests to verify if your code runs as you want. Indeed it is not because your code compile that it exactly does what you want :

- Some errors can rarely appear (particular values, part of code not often cover)
- The retun value can not be right
- ...

7.1 How to test ?

The idea is to write unit tests to execute and cover maximum of the source code.
So, you have to create simple examples which aim at verifying each part of your code runs well and limite cases are well managed. It can seem useless, but it always detects frequent little errors.

For one function, we can have many tests which verify different "critical" cases, so take care of :

- Boundary value
- How managing redundant value
- Coherence of return value
- Random processing
- ...

In **ParadisEO**, test classes are in the *test* directorie. In our case, we put all test classes in *MyProject/test*. A test class is a ".cpp" file containing a "main". For more readability, we recommand you to name your test class with a prefix "t-" of the class you want to test (ex : *MyfileToTest.cpp*(*.h*) / *t-MyfileToTest.cpp*). Remember it is necessary that all procedures of your class integrate different tests for all "critical" cases.

To test, simply use "cassert" library :

```
#include <cassert>
```

And verify return value after a processing :

```
assert(returnValue==expectedValue);
```

Note : returnValue can be a complex type and so it can be done many asserts on it.

To add tests in build process, fill the file *MyProject/test/CMakeLists.txt* :

- Include source directories you need (copy corresponding lines)
- See the test classes in a list (the first instruction foreach adds the extension ".cpp" at each member of the line and the second one adds all listed tests in the test suite) and link libraries

```
include_directories(${CMAKE_SOURCE_DIR}/src)
set(TEST_LIST
t-file1
t-file2
...
t-fileN
)


foreach (test ${TEST_LIST})
set("T_${test}_SOURCES" "${test}.cpp")
add_executable(${test} ${T_${test}_SOURCES})
add_test(${test} ${test})
target_link_libraries(${test} ${PARADISEO_LIBRARIES} yourlib)
endforeach (test)
```

Now you can use **ctest** (cf the end of document) to run test suite.

8 Example

Finally, you can give a simple application using **ParadisEO** and your works :
- Put a ".cpp" file and if necessary, a parameters file in *MyProject/application*
- Fill the *CMakeLists.txt*, you can refer to Example/application)
- Fill the script run.sh to simply run your application

In the CMakeLists.txt file, you have an example to copy parameter file in the build directory by using the following command :

```
>make install
```

9 Use CMake and CTest on Unix

Here we explain how to use **cmake** and **ctest** in order to send report containing different information (build, test, coverage, memory leak).

First, to be able to verify memory leaks, you have to install **valgrind**.

Then, recompile **ParadisEO** in debug mode, if it is not the case (see tutorial on **ParadisEO** website).

In MyProject/build :

```
>rm -rf *
>cmake .. -DCMAKE_BUILD_TYPE=Debug # To be in debug mode
>make
>make doc # To generate documentation
```

Now you can run **ctest** :

```
>ctest -D ExperimentalStart -D ExperimentalTest -D ExperimentalMemCheck -D ExperimentalCoverage -D ExperimentalSubmit
```

Reports are available on http ://cdash.inria.fr/CDash/index.php ?project=ParadisEO