SALOME Kernel resources for developer

Antoine Yessayan, Paul Rascle

Version 0.2 January 28, 2005

This document describes the development environment for C++ and Python. Makefiles generation and usage are introduced in another document: "using the SALOME configuration and building system environment". Development environment is intended here as: trace and debug macros usage; SALOME exceptions usage, in C++ and Python; user CORBA exceptions usage, in C++ and Python, with and without Graphical User Interface; some general purpose services such as singleton, used for CORBA connection and disconnection.

Contents

1 Trace and debug Utilities

During the development process, an execution log is useful to identify problems. This log contains messages, variables values, source files names and line numbers. It is recommended to verify assertions on variables values and if necessary, to stop the execution at debug time, in order to validate all parts of code.

1.1 Two modes: debug and release

The goal of debug mode is to check as many features as possible during the early stages of the development process. The purpose of the utilities provided in SALOME is to help the developer to add detailed traces and check variables values, without writing a lot of code.

When the code is assumed to be valid, the release mode optimizes execution, in terms of speed, memory, and display only user level messages.

But, some informations must always be displayed in both modes: especially messages concerning environment or internal errors, with version identification. When an end user is confronted to such a message, he may refer to a configuration documentation or send the message to the people in charge of SALOME installation, or to the development team, following the kind of error.

1.2 C++ Macros for trace and debug

SALOME provides C++ macros for trace and debug. These macros are in `SALOME/src/SALOMELocalTrace/utilities.h` and this file must be included in C++ source. Some macros are activated only in debug mode, others are always activated. To activate the debug mode, `_DEBUG_` must be defined, which is the case when SALOME Makefiles are generated from configure, without options. When `_DEBUG_` is undefined (release mode: `configure -disable-debug -enable-production`), the debug mode macros are defined empty (they do nothing). So, when switching from debug to release, it is possible (and recommended) to let the macro calls unchanged in the source.

All the macros generate trace messages, stored in a circular buffer pool. A separate thread reads the messages in the buffer pool, and, depending on options given at SALOME start, writes the messages on the standard output, a file, or send them via CORBA, in case of a multi machine configuration.

Three informations are systematically added in front of the information displayed:

- the thread number from which the message come from;
- the name of the source file in which the macros is set;
- the line number of the source file at which the macro is set.

1.2.1 Macros defined in debug and release modes

INFOS_COMPILATION The C++ macro `INFOS_COMPILATION` writes on the trace buffer pool informations about the compiling process:

- the name of the compiler : `g++, KCC, CC, pgCC`;
- the date and the time of the compiling processing process.

This macro `INFOS_COMPILATION` does not have any argument. Moreover, it is defined in both compiling mode : `_DEBUG_` and `_RELEASE_`.

Example :

```
#include "utilities.h"
int main(int argc , char **argv)
{
  INFOS_COMPILATION;
  ...
}
```

INFOS(str) In both compiling mode `_DEBUG_` and `_RELEASE_`, The C++ macro `INFOS` writes on the trace buffer pool the string which has been passed in argument by the user.

Example :

```
#include "utilities.h"
int main(int argc , char **argv)
{
  ...
  INFOS("NORMAL END OF THE PROCESS");
  return 0;
}
```

displays :

```
main.cxx [5] : NORMAL END OF THE PROCESS
```

INTERRUPTION(str) In both compiling mode `_DEBUG_` and `_RELEASE_`, The C++ macro `INTERRUPTION` writes on the trace buffer pool the string, with a special `ABORT` type. When the thread in charge of collecting messages finds this message, it terminates the application, after message treatment.

IMMEDIATE_ABORT(str) In both compiling mode `_DEBUG_` and `_RELEASE_`, The C++ macro `IMMEDIATE_ABORT` writes the message immediately on standard error and exits the application. Remaining messages not treated by the message collector thread are lost.

1.2.2 Macros defined only in debug mode

MESSAGE(str) In `_DEBUG_` compiling mode only, the C++ macro `MESSAGE` writes on the trace buffer pool the string which has been passed in argument by the user. In `_RELEASE_` compiling mode, this macro is blank.

Example :

```
#include "utilities.h"
#include <string>
using namespace std;
int main(int argc , char **argv)
{
```

```
  ...
  const char *str = "Salome";
  MESSAGE(str);
  ... const string st;
  st = "Aster";
  MESSAGE(c_str(st+" and CASTEM"));
  return 0;
}
```

displays :

```
- Trace main.cxx [8] : Salome
- Trace main.cxx [12] : Aster and CASTEM
```

BEGIN_OF(func_name) In `_DEBUG_` compiling mode, The C++ macro `BEGIN_OF` appends the string `"Begin of "` to the one passed in argument by the user and displays the result on the trace buffer pool. In `_RELEASE_` compiling mode, this macro is blank.

Example :

```
#include "utilities.h"
int main(int argc , char **argv)
{
  BEGIN_OF(argv[0]);
  return 0;
}
```

displays :

```
- Trace main.cxx [3] : Begin of a.out
```

END_OF(func_name) In `_DEBUG_` compiling mode, The C++ macro `END_OF` appends the string `"Normal end of "` to the one passed in argument by the user and displays the result on the trace buffer pool. In `_RELEASE_` compiling mode, this macro is blank.

Example :

```
#include "utilities.h"
int main(int argc , char **argv)
{
  END_OF(argv[0]);
  return 0;
}
```

displays :

```
- Trace main.cxx [4] : Normal end of a.out
```

SCRUTE(var) In `_DEBUG_` compiling mode, The C++ macro `SCRUTE` displays its argument which is an application variable followed by the value of the variable. In `_RELEASE_` compiling mode, this macro is blank.

Example :

```
#include "utilities.h"
int main(int argc , char **argv)
{
  const int i=999;
  if( i > 0 ) SCRUTE(i) ; i=i+1;
  return 0;
}
```

displays :

```
- Trace main.cxx [5] : i=999
```

ASSERT(condition) In `_DEBUG_` compiling mode only, The C++ macro `ASSERT` checks the expression passed in argument to be not NULL. If it is NULL the condition is written with the macro `INTERRUPTION` (see above). The process exits after trace of this last message. In `_RELEASE_` compiling mode, this macro is blank. N.B. : if `ASSERT` is already defined, this macro is ignored.

Example :

```
#include "utilities.h"
...
const char *ptrS = fonc();
ASSERT(ptrS!=NULL);
cout << strlen(ptrS);
float table[10];
int k;
...
ASSERT(k<10);
cout << table[k];
```

2 Exceptions

2.1 C++ exceptions: class SALOME_Exception

2.1.1 definition

The class `SALOME_Exception` provides a generic method to send a message, with optional source file name and line number. This class is intended to serve as a base class for all kinds of exceptions SALOME code. All the exceptions derived from `SALOME_Exception` could be handled in a single catch, in which the message associated to the exception is displayed, or sent to a log file.

The class `SALOME_Exception` inherits its behavior from the STL class exception.

2.1.2 usage

The header `SALOME/src/utils/utils_SALOME_Exception.hxx` must be included in the C++ source, when raised or trapped:

```
#include "utils_SALOME_Exception.hxx"
```

The `SALOME_Exception` constructor is:

```
SALOME_Exception( const char *text,
                  const char *fileName=0,
                  const unsigned int lineNumber=0 );
```

The exception is raised like this:

```
throw SALOME_Exception("my pertinent message");
```

or like this:

```
throw SALOME_Exception(LOCALIZED("my pertinent message"));
```

where LOCALIZED is a macro provided with `utils_SALOME_Exception.hxx` which gives file name and line number.

The exception is handled like this:

```
try
  {
    ...
  }
catch (const SALOME_Exception &ex)
  {
    cerr << ex.what() <<endl;
  }
```

The `what()` method overrides the one defined in the STL exception class.

2.2 CORBA exceptions

2.2.1 definition

The idl `SALOME_Exception` provides a generic CORBA exception for SALOME, with an attribute that gives an exception type,a message, plus optional source file name and line number.

This idl is intended to serve for all user CORBA exceptions raised in SALOME code, as IDL specification does not support exception inheritance. So, all the user CORBA exceptions from SALOME could be handled in a single catch.

The exception types defined in idl are:

COMM

CORBA communication problem,

BAD_PARAM

Bad User parameters,

INTERNAL_ERROR

application level problem (often irrecoverable).

CORBA system and user exceptions already defined in the packages used within SALOME, such as OmniORB exceptions, must be handled separately.

2.2.2 usage

CORBA servant, C++ The CORBA Server header for `SALOME_Exception` and a macro to throw the exception are provided with the header `SALOME/src/Utils/Utils_CorbaException.hxx`:

```
#include "Utils_CorbaException.hxx"
```

The exception is raised with a macro which appends file name and line number.

```
if (myStudyName.size() == 0)
   THROW_SALOME_CORBA_EXCEPTION("No Study Name given", \
                                SALOME::BAD_PARAM);
```

CORBA Client, GUI Qt C++ The CORBA Client header for `SALOME_Exception` and a Qt function header that displays a message box are provided in `SALOME/src/SALOMEGUI/SALOMEGUI_QtCatchCorbaException.hxx`:

```
#include "SALOMEGUI_QtCatchCorbaException.hxx"
```

A typical exchange with a CORBA Servant will be:

```
try
  {
    ... // one ore more CORBA calls
  }
catch (const SALOME::SALOME_Exception & S_ex)
  {
    QtCatchCorbaException(S_ex);
  }
```

CORBA Client, C++, without GUI Nothing specific has been provided to the developer yet. See the idl or the Qt function `SALOMEGUI_QtCatchCorbaException.hxx` to see how to get the information given by the exception object.

3 Miscellaneous tools

3.1 Singleton

3.1.1 Definition

A singleton is an application data which is created and deleted only once at the end of the application process. The C++ compiler allows the user to create a static singleton data before the first executable statement. They are deleted after the last statement execution.

The `SINGLETON_` template class deals with dynamic singleton. It is useful for functor objects. For example, an object that connects the application to a system at creation and disconnects the application at deletion.

3.1.2 Usage

To create a single instance a POINT object :

```
# include "Utils_SINGLETON.hxx"
...
POINT *ptrPoint=SINGLETON_<POINT>::Instance() ;
assert(ptrPoint!=NULL) ;
```

No need to delete ptrPoint. Deletion is achieved automatically at exit. If the user tries to create more than one singleton by using the class method `SINGLETON_<TYPE>::Instance()`, the pointer is returned with the same value even if this is done in different functions (threads ?).

```
POINT *p1=SINGLETON_<POINT>::Instance() ;
...
POINT *p2=SINGLETON_<POINT>::Instance() ;
assert(p1==p2)
```

3.1.3 Design description

Here are the principles features of the singleton design :

- the user creates an object of class `TYPE` by using the class method `SINGLETON_<TYPE>::Instance()` which returns a pointer to the single object ;
- to create an object, `SINGLETON_<TYPE>::Instance()` uses the default constructor of class `TYPE` ;
- at the same time, this class method creates a destructor object which is added to the generic list of destructor objects to be executed at the end of the application (`atexit`) ;
- at the end of the application process all the deletions are performed by the `Nettoyage()` C function which executes the destruction objects end then deletes the destructions objects themselves ;
- the `Nettoyage()` C function using atexit() C function is embedded in a static single object `ATEXIT_()`.