Using the LEDA library in non-C++ programming languages

LEDA's functionality can be used in other programming languages. This document explains how the LEDA library should be connected to your code in order to avoid common problems.

Link convention

The data types, functions, and methods inside the LEDA library follow the C++ link convention. This means that you can only link directly to this library if you program in C++. Most other programming languages (for example Java, Delphi and many others) use C linkage convention when linking against other libraries. Thus, you have to encapsulate your calls to LEDA in so-called wrapper functions, which have to use C linkage convention.

C linkage convention is ensured by adding *extern "C"* to the function declaration.

Example:

```
extern "C" void my_function()
{
        << your encapsulated LEDA code goes here >>
}
```

All these functions with C linkage convention have to be placed into a separated file. From now on, we will call this file *LedaLibWrapper.cpp*
This file should then be compiled by a C++ compiler; we will build a small library (let's call it *libLEDAwrapper.a* resp. *libLEDAwrapp.dll*) that contains all the wrapper functions.
You can now use this small wrapper library to write projects in other programming languages. A requirement is of course that your programming language can link against libraries with C linkage convention.

Catching LEDA error messages

Per default, LEDA sends its error messages to the standard error stream. However, the library can be configured to throw exceptions instead. To do so, place the following function into file *LedaLibWrapper.cpp*:

```
#include <LEDA/system/error.h>

extern "C" void InitLedaExceptions()
{
    leda::set_error_handler(leda::exception_error_handler);
}
```

Your code should call the function *InitLedaExceptions()* once. The encapsulated calls to the LEDA library in *LedaLibWrapper.cpp* should be modified like this:

```
extern "C" void my_function()
{
```

```
        Try {
                << your encapsulated LEDA code goes here >>
        }
        catch(leda::leda_exception e)
        {
                << place code to handle the exception, the exception message can be extracted
                   by a call to e.get_msg() >>
        }
}
```

Memory problems and Memory Consumption

LEDA uses its own memory manager to allocate and free memory. Memory is only returned to the operating system at time of destruction of the memory manager or when your unload the LEDA library. There are cases where this may lead to a memory shortage. To avoid these situations, you can force LEDA's memory manager to return its unused memory to the operating system. This is a costly operation so that you should only use it in cases where LEDA allocates large amounts of memory.

Example:

```
#include <LEDA/system/memory.h>

extern "C" void my_memory_consuming_function()
{
        try
        {
                leda::std_memory_mgr.save();
                {
                        << LEDA code that allocates a large amount of memory goes here >>
                }
                leda::std_memory_mgr.restore();
        }
        catch(leda::leda_exception e)
        {
                …
        }
}
```

The memory consuming code is encapsulated between a call to *leda::std_memory_mgr.save()* and *leda::std_memory_mgr.restore()*. These two functions ensure that memory is returned to the operating system. The LEDA code is encapsulated into brackes ({ and }) to ensure that all necessary destructors of your used LEDA data types are called. If you have allocated memory by your own, you have of course to free it by your own.

It is possible to nest the calls to *leda::std_memory_mgr.save()* and *leda::std_memory_mgr.restore*:

```
leda::std_memory_mgr.save();
{
        …
        …
        leda::std_memory_mgr.save();
        {
                …
                …
        }
        leda::std_memory_mgr.restore();
        …
        …
}
leda::std_memory_mgr.restore();
```

If you still believe that LEDA's memory manager leads to problems with your code contact Algorithmic Solutions and ask for a version with LEDA's memory management being disabled.