Matrix linking.

'Sharp as C' continued.

Intro

Everyone knows what static and dynamic linking is. There would be a waste of time to delve either into their comparison or description once again. The basic problem with standard linkage is that in case the bug is found, we need to rebuild the code and restart the system. Having repeated again and again these simple procedures, it has become one of most boring and, perhaps also the basic time-eaters during the developers working hours. The idea of matrix linking is intended to annihilate these steps, which are recompilation and/or restarting the system.

The idea, the theory.

Let me put some theory in the beginning, it will be not so simple and funny as a practice, yet putting the practice before the theory might have look strange. Having this in mind there is a reason to skip this clause and go directly to the next one, leaving this for the more leisure time of yours.

In theory, abstract theory, let me say we have some source code, implemented by means of some algorithmic language. Let us call this source code an algorithm. Let those modules which we will have as result of compilation of our source code (an algorithm) we will be called “basic modules”. These “basic modules” are program modules. These modules are to be loadable at run-time.

What I’m trying to say is to describe in general terms of the application based on C language. In another words, the all we have is the C-source code which after compilation becomes shared libraries. Actually, that’s it.

Also let me invent one addition program module; let it be called “dispatching” one. This dispatching module will have two-sized matrix of pointers. Let us say the amount of rows in this matrix will be equal to amount of “basic” modules, and each particular row will be corresponded to the “basic” module in such wise that amount of columns in this row will be equal to the amount of functions in corresponded module, and, as you might guess each particular element in this row will be corresponded to the function of the module (it will be described a little bit lately in details).

The so-called “basic” module has a pecularity, or rather to say that peculiarity has its compiled code. Let us consider a sample:

<table>
<tr><td>Line #</td><td>Lexeme</td><td colspan="2">Comments</td></tr>
<tr><td>1</td><td>Declare Function1</td><td colspan="2">Declaration of function1</td></tr>
<tr><td>2</td><td>Begin</td><td colspan="2" rowspan="2">Body of function1</td></tr>
<tr><td>3</td><td>End</td></tr>
<tr><td>4</td><td></td><td colspan="2"></td></tr>
<tr><td>5</td><td>Declare Function2</td><td colspan="2">Declaration of function2</td></tr>
<tr><td>6</td><td>Begin</td><td></td><td rowspan="3">Body of function2</td></tr>
<tr><td>7</td><td>Call Function1</td><td>Call for function1</td></tr>
<tr><td>8</td><td>End</td><td></td></tr>
</table>

Diagram 1

(c) George Shagov, 2005
December 1, 2005

On the diagram 1 we have an abstract source code (algorithm), written in abstract language. The algorithm represents two functions function1 and function2, and function2 has a call to a function1.

Let us say this source code of some basic module and this module has a number M, which corresponds to row M in matrix. The peculiarity of the compiled code of such module will be that the call for a function2 inside the function1 will not be compiled as a direct call to a function1, yet instead of that it will be compiled as a call to a function which is placed in row M column 1 of the matrix (in case function1 corresponds to a column 1).

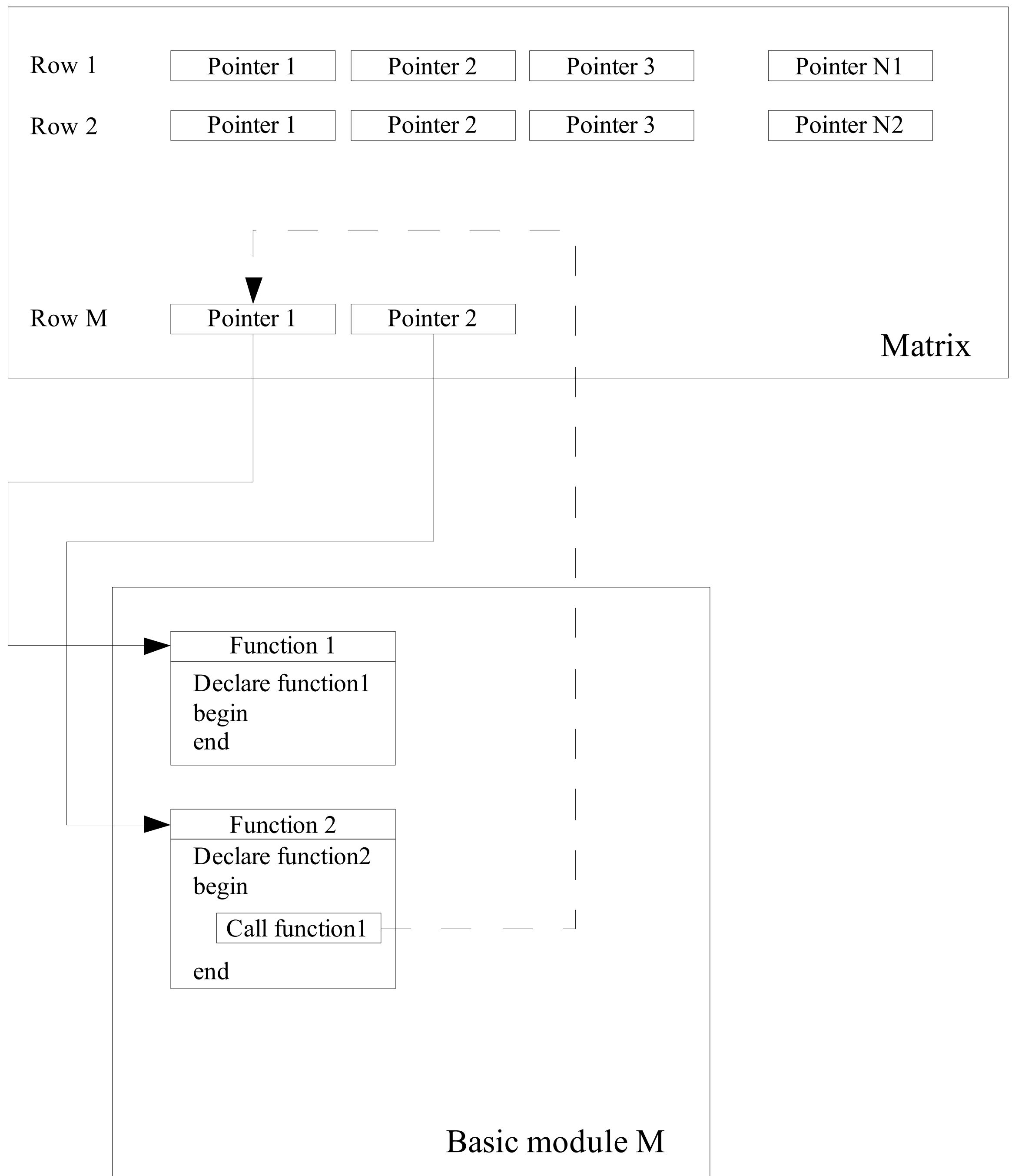

figure 1.

Let us consider the figure 1.

The solid line means that the value of the element in the matrix points to a corresponded function in the corresponded module, for instance Pointer 1 in Row M has a value of function 1 of module M.

The dotted line means that the call of a specific function will be ‘translated’ to a call of a corresponded function the pointer of which is placed on a corresponded position in matrix, for instance: the call for a function1 of module M will be realized as a call to a function whose pointer is placed in Row M column 1 of a matrix.

Perhaps there is one more thing is to be mentioned before we start with practice. The thing is the so-called “proxy-module”. Let me say we have the program module similar to “basic” one, (the same stuff of functions) with that difference that the functions of the “proxy module” are not implementing the algorithm, but rather make a decision about what “basic” module to be used in order to run such or other function or, it might be, what particular script-file, implementing such or other function of the algorithm, to be executed, using a script-machine, let us consider the figure 2:

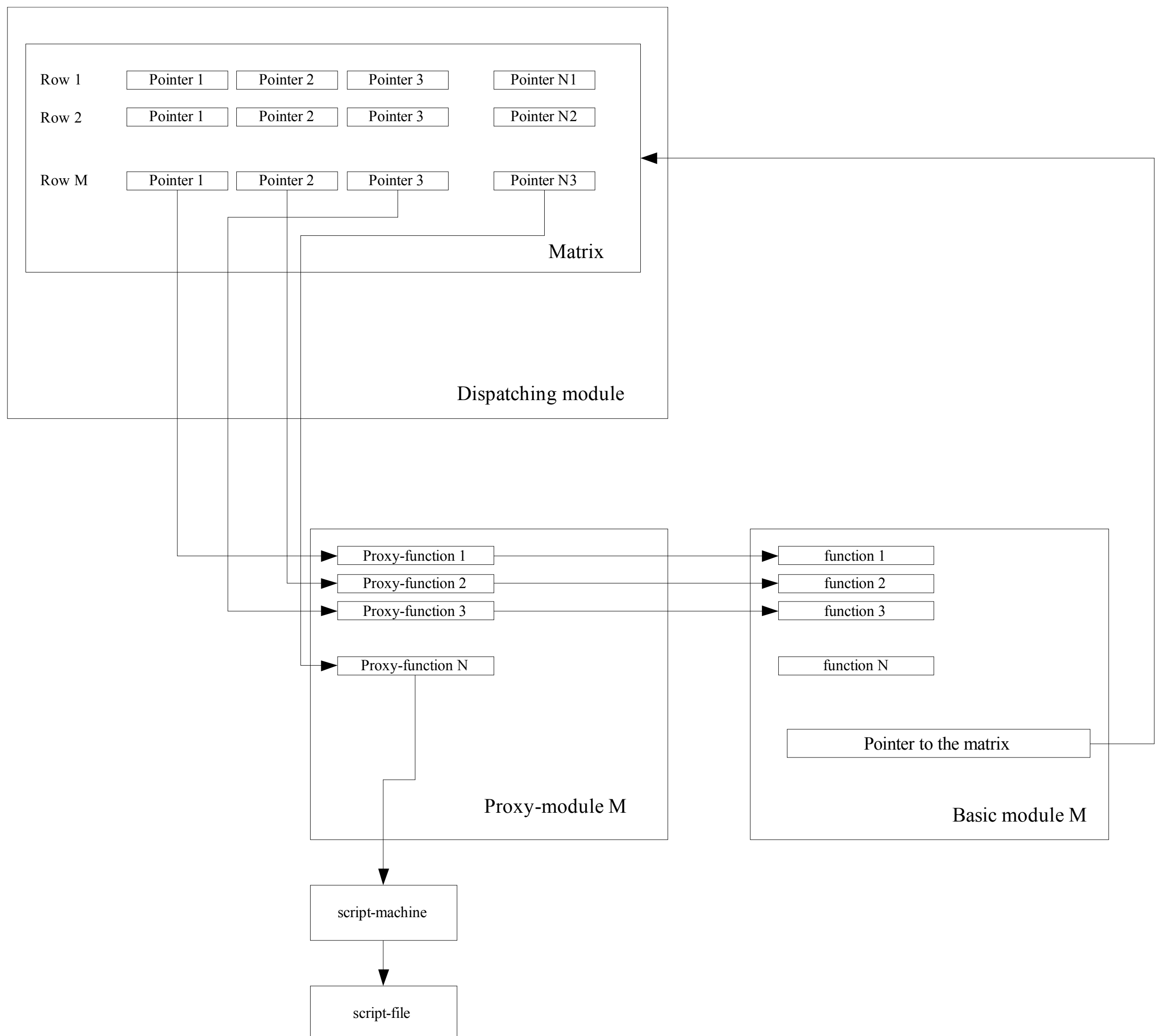

figure 2.

The algorithm of a proxy function in this sample looks pretty simple, videlicet: in case the script-file is found it’s going to be executed, using the script-machine, if not – the corresponded function from a corresponded basic module will be executed.

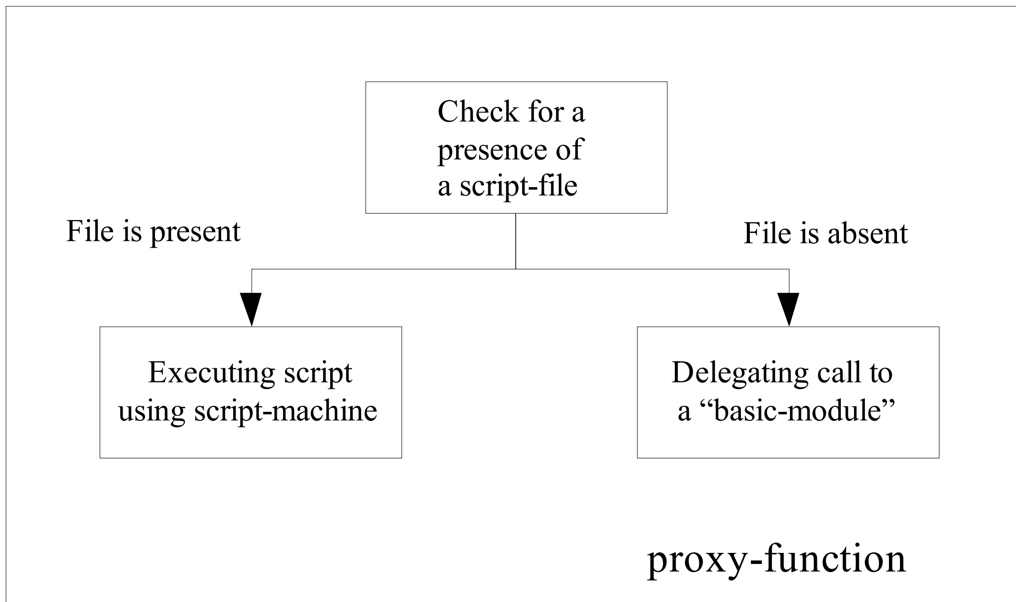

figure 3

The practice.

Just one, but very important limitation:
The source code is implemented in plain C.
This section will show how it works. (The sample code is written for Microsoft windows platform)

This is an application tree:

```
├───common.gnrtd
├───c_dispatcher
├───c_parser
├───frontend_app
├───include
├───module0
│   ├───cint_module0_proxy
│   └───module0_proxy
└───output
```

Inside folder fronend_app the main (console) application is located. There is only one file there: frontend_app.cpp

```
/** frontend_app.cpp
 *  (c) George Shagov, 2005
 */
#include "./../include/os.h"
#include <stdio.h>
#include <ctype.h>
#include "./../include/my_structs.h"
#include "./../common.gnrtd/script01.fntypes.gnrtd.h"
#include "G__ci.h"

typedef int (__cdecl *D_EXECUTE)();
typedef HINSTANCE (__cdecl *D_GETMODULE)(const char*);
static char* s_sUsage = "Usage:\ntype 'd' to execute dispatching script\ntype 'e' to execute the entry
point.\ntype 'x' or 'q' to exit\n";

int main(int argc, char* argv[])
{
  /*
   *     declarations
   */
  char c = ' ';
  HINSTANCE hInstanceEntry = NULL;
  c__my_entry_point_type pEntry = NULL;
  char sDispatherPath[128];
  HINSTANCE hInstanceDisp = NULL;
  D_EXECUTE pExecute = NULL;
  D_GETMODULE pGetModule = NULL;
  SMyStructure myStruct;
  char sMyString[32];

  /*
   *     getting dispatching module
   *   and its entry-points
   */
  sprintf(sDispatherPath, "%sc_dispatcher.dll", PATH_TO_OUTPUT);
  hInstanceDisp = LOADLIBRARY(sDispatherPath, RTLD_NOW);
  pExecute = (D_EXECUTE)GETPROCADDRESS(hInstanceDisp, "g_Execute");
  pGetModule = (D_GETMODULE)GETPROCADDRESS(hInstanceDisp, "g_GetModule");
```

```
/** frontend_app.cpp
 *  (c) George Shagov, 2005
 * continued...
 */

  /*
   *     main loop
   */
  while (c != 'x' && c != 'q')
  {
    /*
    *   initial data for the script
    */
    strcpy(sMyString, "My string here.");
    myStruct.m_nVal = 0;
    strcpy(myStruct.m_sString, "initial");
    switch(c)
    {
    case 'd':
      /*
       *          loading "basic" module here
       */
      printf("dispatching...\n");
      pExecute();
        break;
    case 'e':
      /*
       *          executing the script
       *   using loaded "basic" module
       */
      {
        printf("executing\n");
        G__init_cint(CINT_COMMAND_STRING);
        hInstanceEntry = pGetModule("module0");
        pEntry = (c__my_entry_point_type)GETPROCADDRESS(hInstanceEntry,
"c__my_entry_point_impl");
        pEntry(argc, sMyString, &myStruct);
        G__scratch_all(); /* Clean up Cint */
      }
        break;
    default:
      break;
    }
    printf(s_sUsage);
    c=getchar();
  }
        return 0;
}
```

As you can see here there are two procedures here:

1. The dispatching procedure. The realization of which is placed into the c_dispatcher module. And the basic idea is that the dispatcher loads the module we need. (case ‘d’)
2. The second procedure is executing the algorithm itself. (case ‘e’)

Here is the algorithm (script) which is placed in module0 folder:

```
/** script01.c_
 * (c) George Shagov, 2005
 */
int c__get_value_1_impl(char* pString)
{
  return 1;
}

int c__get_value_2_impl(int nArg)
{
  return 2;
}

int c__call_in_case_varables_are_equal_impl(SMyStructure* pMyStruct)
{
  pMyStruct->m_nVal = 0;
  strcpy(pMyStruct->m_sString, "equal");
  return 0;
}

int c__call_in_case_varables_are_not_equal_impl(SMyStructure* pMyStruct)
{
  pMyStruct->m_nVal = 0;
  strcpy(pMyStruct->m_sString, "not equal");
  return 0;
}

int c__re_entry_impl(int nArg, char* pString, SMyStructure* pMyStruct)
{
  int nVar1 = c__get_value_1(pString);
  int nVar2 = c__get_value_2(nArg);

  if (nVar1 == nVar2)
  {
    c__call_in_case_varables_are_equal(pMyStruct);
  }
  else
  {
    c__call_in_case_varables_are_not_equal(pMyStruct);
  }

  return 11;
}

int c__my_entry_point_impl(int nArg, char* pString, SMyStructure* pMyStruct)
{
  int nRet;

  printf("-----------\nbefore:\n");
  printf("nArg: %d, string: %s\n", nArg, pString);
  printf("pMyStruct->m_nVal: %d, pMyStruct->m_sString: %s\n", pMyStruct->m_nVal, pMyStruct-
>m_sString);

  nRet = c__re_entry(nArg, pString, pMyStruct);

  printf("++++++after:\n");
  printf("nArg: %d, string: %s\n", nArg, pString);
  printf("pMyStruct->m_nVal: %d, pMyStruct->m_sString: %s\n", pMyStruct->m_nVal, pMyStruct-
>m_sString);
  printf("ret: %d\n------------\n", nRet);

  return nRet;
}
```

One more thing to be shown:

```
/** my_structs.h
 * (c) George Shagov, 2005
 */

#ifndef __MY_STRUCTS_H__
#define __MY_STRUCTS_H__


typedef struct SMyStructure
{
  int m_nVal;
  char m_sString[16];
} SMyStructure;

#endif /* __MY_STRUCTS_H__ */
```

c__my_entry_point_impl is an entry point to be called from frontend_app. Script01.gnrtd.c is the mere copy of the original script. Script01.gnrtd.h represents the declarations.

To complete the idea there needs to be some additional code to be shown:

```
#define c__get_value_1 c__get_value_1_stub
#define c__get_value_2 c__get_value_2_stub
#define c__call_in_case_varables_are_equal c__call_in_case_varables_are_equal_stub
#define c__call_in_case_varables_are_not_equal c__call_in_case_varables_are_not_equal_stub
#define c__re_entry c__re_entry_stub
#define c__my_entry_point c__my_entry_point_stub
```

What these _stub functions are, it will be explained a little bit lately.

Now, getting back to the theory. Let me say that the C-code script here is a matrix, with atomic element - a function. This point of view says that calls inside the script should not go directly to its implementation but rather should go through that element which is placed at corresponded position in matrix. It might look complex at the first look, yet it gives us exactly that flexibility we are looking after. That approach says also that any particular element in matrix might be substituted to any other without restarting the system. All the calls to this element will go through the newly ‘loaded’ functionality.

This is the basic principle.

Let us see how it works.

Each particular module, realizing script functionality, should also provide additional entry points in order to identify itself and instantiate the matrix.

The matrix looks pretty simple, like this:

```
typedef struct S_FnTable
{
  void* _pTable[C__MAX_FUNCTIONS];
}
S_FnTable;

typedef S_FnTable Matrix[C__MAX_MODULES];
```

These additional functionality, by means of which each particular ‘basic’ module to be extended might look like these:

```
/** module0.dll.c
 *  (c) George Shagov, 2005
 */

#include "./../include/os.h"
#include "./../include/c_fn_s.h"
#include "module0.dll.h "

S_FnTable g_pFnTables[C__MAX_MODULES];

static int g_nModuleID = 0;

int g_GetModuleID()
{
   return g_nModuleID;
}

int g_SetFnTable(int nModuleID, const S_FnTable* pTable)
{
   memcpy(g_pFnTables[nModuleID]._pTable, pTable, sizeof(S_FnTable));
   return 0;
}
```

And also the code, which fulfills the matrix, or rather to say one raw of it:

```
/** script01.fntable.c
 * (c) George Shagov, 2005
 */

/**********************************************************************
 *
 * this file is automatically generated from script01.c_
 * do not mofify it
 *
 **********************************************************************/

#include "./../include/os.h"
#include "./../include/my_structs.h"
#include "./../common.gnrtd/script01.fntypes.gnrtd.h"
#include "./../include/c_fn_s.h"
#include "./script01.gnrtd.h"
#include "module0.dll.h"

int g_GetFnTable(S_FnTable* pTable)
{
  void* pProc = NULL;
  pTable->_pTable[c__get_value_1_ID] = (void*)c__get_value_1_impl;
  pTable->_pTable[c__get_value_2_ID] = (void*)c__get_value_2_impl;
  pTable->_pTable[c__call_in_case_varables_are_equal_ID] = (void*)
c__call_in_case_varables_are_equal_impl;
  pTable->_pTable[c__call_in_case_varables_are_not_equal_ID] = (void*)
c__call_in_case_varables_are_not_equal_impl;
  pTable->_pTable[c__re_entry_ID] = (void*)c__re_entry_impl;
  pTable->_pTable[c__my_entry_point_ID] = (void*)c__my_entry_point_impl;

  return 0;
}
```

As you can see each particular module ‘knows’ its id, able to retrieve it and also has a functionality to fill up its own matrix.

Let us take a look at functionality of c_dispatcher:

```
/** c_dispatcher.cpp
 *  (c) George Shagov, 2005
 */

#include <stdio.h>
#include "./../include/os.h"
#include "c_dispatcher.h"
#include "./../include/c_fn_s.h"
#include "./../include/c_modules_fndecl.h"
#include "G__ci.h"          /* Cint header file */

#define D__MAX_MODULES 2

typedef struct SModule
{
   char* _sName;
   HINSTANCE _hInstance;
} SModule;


Matrix* g_pMatrix = NULL;


   extern void G__c_setup(); /* defined in G__clink.c */



SModule s_Modules[D__MAX_MODULES] =
{
   { "module0", NULL },
   { "module1", NULL }
};

int s_GetModuleID(const char* sName)
{
   int i=0;
   for (i=0; i<D__MAX_MODULES; i++)
      if (0 == strcmp(s_Modules[i]._sName, sName))
         return i;
   return -1;
}
```

```
/** c_dispatcher.cpp
 *  (c) George Shagov, 2005
 */

int g_LoadModule(const char* sModule, const char* sPrefix)
{
  char sDll[128];
  HINSTANCE hModule = NULL;
  C__GETMODULEID pModuleID = NULL;
  C__GETFNTABLE pGetFnTable = NULL;
  C__SETMATRIX pSetMatrix = NULL;
  int nModuleID = -1;


  /**
   *   loading module
   */
  sprintf(sDll, "%s%s_%s.dll", PATH_TO_OUTPUT, sModule, sPrefix);
  hModule = LOADLIBRARY(sDll, RTLD_NOW);

  if (!hModule)
  {
    printf("Unable to load library: %s\n", sDll);
    return 1;
  }

  /**
   *   getting entry points from loaded module
   */
  pModuleID = (C__GETMODULEID)GETPROCADDRESS(hModule, "g_GetModuleID");
  pGetFnTable = (C__GETFNTABLE)GETPROCADDRESS(hModule, "g_GetFnTable");
  pSetMatrix = (C__SETMATRIX)GETPROCADDRESS(hModule, "g_SetMatrix");

  nModuleID = pModuleID();

  /*
   *     by this call we are getting function table
   *   of the module
   */
  pGetFnTable(&((*g_pMatrix)[nModuleID]));

  /**
   *   setting up global matrix
   */
  pSetMatrix(g_pMatrix);

  /*
   *     freeing the previous module
   */
  if (s_Modules[nModuleID]._hInstance != NULL)
    FREELIBRARY(s_Modules[nModuleID]._hInstance);

  /*
   *     loading the new one
   */
  s_Modules[nModuleID]._hInstance = hModule;

  return 0;

}
```

```
/** c_dispatcher.cpp
 *  (c) George Shagov, 2005
 */


int g_Execute()
{
  char sExecute[128];
  char sPthToScript[128];
  G__value ret;

  sprintf(sPthToScript, "%s %sc_dispath.script.c", CINT_COMMAND_STRING, PATH_TO_OUTPUT);
  G__init_cint(sPthToScript); /* initialize Cint */
  G__c_setup();

  sprintf(sExecute,"c__execute();");
  ret = G__calc(sExecute); /* Call Cint parser */
  G__scratch_all(); /* Clean up Cint */
  return 0;
}

HINSTANCE g_GetModule(const char* sModule)
{
  int nModuleID = s_GetModuleID(sModule);
  if (-1 == nModuleID)
  {
    printf("invalid name:%s", sModule);
    return 0;
  }
  return s_Modules[nModuleID]._hInstance;
}
```

There is a code related to so-called ‘dispatching’ procedure (g_Execute). This procedure calls to the external script by means of cint. (Cint is free C-interpreter, powerful enough and very suitable for this demo), c-script calls for g_LoadModule.

The code of external script (c_dispatch.script.c):

```
#include <stdio.h>

int c__execute()
{
   printf("c__execute\n");
   g_LoadModule("module0", "stub");
   return 0;
}
```

It is possible to understand by now that at the first step we should load our module, and only after – to execute, which is obvious.

Let us take a look at the original script. It’s easy to see that all declarations are performed using _impl suffix. It’s intentially. Then, what happens after ‘real’ call to the function, which is not suffixed. There happens a call to a so-called stub function. The stub-function looks like this:

```
extern int g_nModuleID;
extern Matrix* g_pGlobalMatrix;

int c__get_value_1_stub( char* pString)
{
  c__get_value_1_type pFn = (c__get_value_1_type)(*g_pGlobalMatrix)[module0_ID]._pTable
[c__get_value_1_ID];
  printf("c__get_value_1_stub\n");
  return pFn(pString);
}
```

As you can see here actual call is delegated to a function which pointer is in the matrix. Have you got a trick? Simple, isn't it?

So, it means we can substitute any matrix's element (which is pointer to a function) to whatsoever (and whenever) we would like to.

Let us see how it works now:

The context of c_\output folder (after getting the project built) looks like this:

```
c_dispatcher.dll
c_dispath.script.c
frontend_app.exe
module0_stub.dll
```

And the code of external script (c_dispatch.script.c):

```
#include <stdio.h>

int c__execute()
{
   printf("c__execute\n");
   g_LoadModule("module0", "stub");
   return 0;
}
```

It means we will load a stub module.

Module0_stub.dll is the compiled module of our script.

Starting the application and typing 'd' (dispatching) we got:

```
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
d
dispatching...
c__execute
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
```

It means out module is loaded, executing:

```
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
e
executing
-----------
before:
nArg: 1, string: My string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: initial
c__re_entry_stub
c__get_value_1_stub
c__get_value_2_stub
c__call_in_case_varables_are_not_equal_stub
++++++after:
nArg: 1, string: My string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: not equal
ret: 11
-------------
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
```

As we can see it works. We have our calls going through the matrix, as we might see stub's output.

Let us here introduce new entity, let it has a name 'proxy-module'. This is a separate module, which will export exactly the same functions which 'basic' module does, yet these functions will do nothing but delegate the call to 'basic' module.

Let us take a look at the proxy-function:

```
static int s_IsFunctionOverloaded(const char* sFnName)
{
  char sFile[128];
  FILE* f = NULL;
  sprintf(sFile, "%s%s.c", PATH_TO_OUTPUT, sFnName);
  f = fopen(sFile, "r");
  if (!f)
    return 0;
  fclose(f);
  return 1;
}


int c__get_value_1_impl( char* pString)
{
  if (s_IsFunctionOverloaded("c__get_value_1")) {
    char tmp[128];
    char sPath[128];
    int nRet;

    sprintf(sPath, "%sc__get_value_1.c", PATH_TO_OUTPUT);
    G__loadfile(sPath); /* initialize Cint */
    printf("proxy: c__get_value_1_impl -- cint\n");
    sprintf(tmp,"c__get_value_1_impl((void*)0x%08p);", (void*)pString);
    nRet = G__calc(tmp).obj.i; /* Call Cint parser */
    G__unloadfile(sPath); /* initialize Cint */
    return nRet;
  } else {
    c__get_value_1_type pFn = (c__get_value_1_type)GETPROCADDRESS(s_hStubModule,
"c__get_value_1_impl");
    printf("proxy: c__get_value_1_impl\n");
    return pFn(pString);
  }
}
```

As you can see at the first it checks for a file: <function_name>.c if it is present it calls for cint to execute it, no – executes the corresponded function from a basic module. That's it.

The context of c_\output folder (after getting the project built) looks like this:

```
c_dispatcher.dll
c_dispath.script.c
cint_module0_proxy.dll
frontend_app.exe
module0_stub.dll
module0_proxy.dll
```

At the first we should change c_dispath.script.c file, it should look like this:

```
#include <stdio.h>

int c__execute()
{
   printf("c__execute\n");
   g_LoadModule("module0", "proxy");
   return 0;
}
```

Reloading the module, typing 'd' in the console, executing, typing 'e':

```
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
d
dispatching...
c__execute
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
e
executing
proxy: c__my_entry_point_impl
-----------
before:
nArg: 1, string: My string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: initial
c__re_entry_stub
proxy: c__re_entry_impl
c__get_value_1_stub
proxy: c__get_value_1_impl
c__get_value_2_stub
proxy: c__get_value_2_impl
c__call_in_case_varables_are_not_equal_stub
proxy: c__call_in_case_varables_are_not_equal_impl
++++++after:
nArg: 1, string: My string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: not equal
ret: 11
-------------
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
```

So, now we are going through the proxy. Let us see how the trick works. In order to do that we should create a file, let it be c__get_value_1.c and put there a functionality of c__get_value_1 function, like this, for instance:

```
// my_script.cpp : Defines the entry point for the DLL application.
//

#include <stdio.h>
#include "..\\include\\my_structs.h"
#pragma include_noerr <cint_module0_proxy.dll>

int c__get_value_1_impl(char* pString)
{
   pString[1] = 'X';
   printf("c__get_value_1 ==>> str: %s\n", pString);
   return 2;
}
```

And the output:

```
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
e
executing
proxy: c__my_entry_point_impl
-----------
before:
nArg: 1, string: My string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: initial
c__re_entry_stub
proxy: c__re_entry_impl
c__get_value_1_stub
proxy: c__get_value_1_impl -- cint
c__get_value_1 ==>> str: MX string here.
c__get_value_2_stub
proxy: c__get_value_2_impl
c__call_in_case_varables_are_equal_stub
proxy: c__call_in_case_varables_are_equal_impl
++++++after:
nArg: 1, string: MX string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: equal
ret: 11
-------------
```

The difference is bolded and underlined.

Let us go some further, let we create a file c__re_entry.c and its content will be:

```
#include "..\\include\\my_structs.h"
#pragma include_noerr <cint_module0_proxy.dll>

int c__re_entry_impl(int nArg, char* pString, SMyStructure* pMyStruct)
{
   printf("\"I'll not be juggled with.\nTo hell, allegiance! Vows, to the
blackest devil!\nConscience and grace, to the profoundest pit!\nI dare
damnation. To this point I stand,\"\n");
   printf("...for this is script\n");

   int nVar1 = c__get_value_1(pString);
   int nVar2 = c__get_value_2(nArg);

   if (nVar1 == nVar2)
   {
      c__call_in_case_varables_are_equal(pMyStruct);
   }
   else
   {
      c__call_in_case_varables_are_not_equal(pMyStruct);
   }

   return 11;
}
```

And the output:

```
Usage:
type 'd' to execute dispatching script
type 'e' to execute the entry point.
type 'x' or 'q' to exit
e
executing
proxy: c__my_entry_point_impl
-----------
before:
nArg: 1, string: My string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: initial
c__re_entry_stub
proxy: c__re_entry_impl -- cint
"I'll not be juggled with.
To hell, allegiance! Vows, to the blackest devil!
Conscience and grace, to the profoundest pit!
I dare damnation. To this point I stand,"
...for this is script
proxy: c__get_value_1_impl -- cint
c__get_value_1 ==>> str: MX string here.
proxy: c__get_value_2_impl
proxy: c__call_in_case_varables_are_equal_impl
++++++after:
nArg: 1, string: MX string here.
pMyStruct->m_nVal: 0, pMyStruct->m_sString: equal
ret: 11
-------------
```

So, the context of output folder by now is this:

```
c__get_value_1.c
c__re_entry.c
c_dispatcher.dll
c_dispath.script.c
cint_module0_proxy.dll
frontend_app.exe
module0_native.dll
module0_proxy.dll
module0_stub.dll
```

Performance.

Yes, of cause, using script instead of native code does mean significant lose of performance, yet there are to things to say

1. In the systems where performance is a key point, such as real-time systems, no substitution to be allowed. It means there should not be any dispatcher library and all the calls to be compiled as direct ones and linked during the compilation. In this approach there will not be any losing of performance. Yet in development in QA where possibility for substation is highly required but performance does not play a significant role this approach will be applicable.
2. The first rule is too strict. As you can see in the output folder there is module_native.dll. What is this? Let us called this library a 'native module'. This is exactly the basic module with that exception that all the 'local' calls (calls inside one module) compiled like a direct ones, no any stub involved. Any calls which go to 'external' module should go through the matrix. IMHO this decision is more than enough in order to get the performance problem rip'd.

Cons and pros.

Had I patience and time I would write a book here, or two.
Yet in brief.

Disadvantages.

- The solving task in general approach, using OOP languages, does look too complicated.
- The build procedure becomes more complicated, additional parsing is required.
- There should be an interpreter supplied
- Read the performance section
- Using C as a script might cause some problems, since C, by default, has a direct access to memory and has no any mechanism of automatic unwinding, which might potentially cause leaks. Yet, that should be a C-script, not a C, it means that all functions which uses access to memory should be exposed as entities.

Benefits

- Ability to change the business logic run-time.
- A control. Just think what we able to do having all entry-points in our hands.
  - Logging
  - Error handling
  - Parameters tracing

Future.

Everyone knows how huge and hardly manageable might be C++ applications. Even very simple change might cost hours. This approach is created to resolve the issue. Yes, of cause, moving the problem from the shoulder of compiler to the interpreter might sound not a best decision, yet, it is obvious not all the code should interpreted (that would be just impossible due to performance), but only those logic which was changed. And this is on the fly, no restarting involved. Obviously such approach has a right to live and I believe will give a lot of benefits to quality of software application development.

Author

(c) George Shagov, 2005
George@georgeshagov.com