Library Loading and Plugins

Marc Dürner

Shared Libraries (Win32)

```
std::basic_string<TCHAR> path = _T(“some/path/to.dll“);

HANDLE h = LoadLibrary( tpath.c_str() );

void* sym = GetProcAddress(h, “SymbolName“);

// use sym

FreeLibrary(h);
```

Shared Libraries (POSIX)

```
std::basic_string<char>  path = “some/path/to.so“;

int h = dlopen( path.c_str(), RTLD_NOW|RTLD_GLOBAL );

void* sym = dlsym(h, “SymbolName“);

// use sym

dlclose(h);
```

Shared Libraries (portable)

```
try
{
  Library shlib(“some/path/to.so“);

  void* sym1 = shlib[“Symbol1“];           // no throw
  if( ! sym1 )
    return ERROR;

  void* sym2 = shlib.getSymbol(“Symbol2“); // throws SymbolNotFound
}
catch(const SymbolNotFound& e)
{
}
catch(const AccessFailed& e)
{
}
```

What‘s the Problem?

What‘s the Problem?

--- RED ALERT --- RED ALERT --- RED ALERT ---

6.3.2.3:8 A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the pointed-to type, the behavior is undefined.

NOTE: cast from void* to function pointer is undefined behaviour

What‘s the Problem?

- Name mangling
  - Use export “C“
- Export resolvable symbols
  - _declspec(export), visibility for gcc, .def files etc…
- Do not rely on platform specific library init/finalisation
  - No init()/fini() etc…

What‘s the Problem?

- Platforms do not allow to load libraries at all
  - iOS, some linux

- Not always possible to load same symbol from different libs

What‘s the Solution?

Example: The Greeter Plugin

- The Greeter Plugin Interface
  - Classic OO design using virtual base classes
  - Plugin Interface defined in header
  - Plugins implement this interface

Greeter.h

```
class Greeter
{
  public:
    virtual ~Greeter()
    {}

    virtual void sayHello() = 0;
};
```

Example: The Greeter Plugin

- The Greeter Plugin Implementation
  - Exports a variable, not functions (also type safe!!!)
  - C Linkage, even though Plugin is a C++ class

EnglishGreeter.cpp (built to plugin.so)

```
class EnglishGreeter : public Greeter
{
  public:
    void sayHello()
    { std::cout << "Hello World"; }
};

static BasicPlugin<EnglishGreeter, Greeter> _enGreeter("en");

extern "C" {
  EXPORT PluginId* PluginList[] = { &_enGreeter, 0 };
}
```

Example: The Greeter Plugin

- The PluginId
  - Allows typesafe cast to derived Plugin<I>
- The Plugin<I>
  - Is a factory interface to create objects implementing I
- The BasicPlugin<T, I>
  - Is a factory to create T implementing I
  - Normal Ctor instead of init/fini

```
PluginId
   + type() : type_info
         ↑
Plugin<I>
   + type() : type_info
   + create() : I*
   + feature() : string
         ↑
BasicPlugin<T, I>
   + create() : I*
   + feature() : string
```

Example: The Greeter Plugin

- The PluginManager
  - Loads an array of PluginId and keeps all which are convertible to Plugin<Greeter>
  - Different symbol names can be resolved for each plugin library
  - Can find Plugin<Greeter> by feature string to create instances

```
PluginManager<Greeter> manager;
manager.loadPlugin("PluginList", "/path/to/plugin.so");

Greeter* greeter = manager.create("en");
if(greeter)
{
  greeter->sayHello();
  manager.destroy(greeter);
}
```

Example: The Greeter Plugin

- Multiple plugins and types in one DLL
- Allows Builtins where plugins aren‘t possible

Builtin Example:

```
BasicPlugin<GermanGreeter, Greeter> deGreeter;

PluginManager<Greeter> manager;
manager.registerPlugin(deGreeter);
```