N2074=06-0144
2006-09-07
Daveed Vandevoorde
daveed@vandevoorde.com

Plugins in C++

1 Introduction

1.1 What is a plugin?

In this paper, a plugin is code that can become part of a program under that program's control. For C and C++ programs plugins are typically implemented as shared libraries that are dynamically loaded (and sometimes unloaded) by the main program. Example applications include media players (e.g., an MPEG4 player) for web browsers, specialized filters for image processing applications (e.g., Adobe Photoshop), dynamically loadable drivers for operating systems (e.g., Linux modules), and so forth.

1.2 Modules and dynamic libraries

At the Berlin meeting in April 2006, at the conclusion of a presentation of the "C++ Modules" proposal (N1964), the Evolution working group gave clear direction that its Modules & Dynamic Libraries subgroup should further pursue the idea of mapping a notion of dynamic libraries on the C++ modules concept presented in N1964.
This paper is an attempt at responding to that request by selecting a specific but important application of dynamic libraries (plugins) and casting it into a natural and portable C++ form that is implementable with (but not inherently tied to) shared library technology.
In this initial revision, the proposal is only a sketch leaving out both formalism and some specific design choices. Details will be worked out if the general direction is found desirable. This paper assumes at least superficial familiarity with N2073 (which is a significant revision of N1964).

1.3 What this proposal is not

This proposal expressly does not attempt to capture all the facilities offered by various operating systems to load dynamic libraries under program control. It also does not attempt to provide a simple transition path from code that currently uses operating-system-specific facilities to standard C++ constructs. Finally, it is not attempting to address any dynamic library applications other than plugins.

1.4 Phases of translation, translation units, etc.

The standard defining the C++ programming language is carefully crafted to avoid being tied to particular implementation technology, while also ensuring that it is efficiently implementable using the most popular technology. Specifically, C++ can be (and usually is) efficiently implemented via native code compilers and symbolic linkers, but it can also practically be realized as an interpreted environment.

Because of this technological neutrality the standard does not talk normatively about object files, compilers, linkers, shared libraries, name mangling, ABIs (application binary interfaces), etc. Instead, it uses the more abstract notions of translation units, phases of translation, instantiation units, the ODR (one-definition rule), "names for linkage purposes", an abstract machine, and so forth. The plugin language support will similarly need to be expressed in terms of these abstract notions, and the proposed approach is compatible with such a constraint.

2 Plugin capabilities

This section looks at a few common capabilities of shared-library-based plugins. The purpose is to identify some underlying mechanisms and to explicitly exclude some capabilities from this proposal. (I.e., the descriptions in this section are not the proposal; just the identification of possible general capabilities.)

2.1 Loading

The minimum capability needed to deal with plugins is an interface that allows the "loading" of a plugin. This amounts to identifying the program units that need to be made part of the program. This identification can happen in various ways but on all known platforms that support some form of plugin mechanism for C and C++ programs the basic underlying identification mechanism is the name of a file (almost always: a dynamic library file).
Loaded plugins can form a dependency graph: One plugin might load another one, and two plugins may both independently load a third.
Plugins may also have dependencies on non-plugin libraries that may or may not already be part of the running application. For example, both the initial program and the plugin may access **`std::cout`**.

2.2 Dynamic entity access

Although loading a plugin could potentially trigger side-effects through the dynamic initialization of its namespace scope variables, it's fair to say that any set of plugin capabilities would be disappointing if there weren't a way to at least call a function or member function in the plugin.
For plugins defined in terms of shared libraries, the identification of the entities in the plugin is through their names as seen by the linker/loader: For C++ that is often a mangled name. A satisfactory standard solution should not rely on the programmer having to find out what the mangled names are. (Curent platform-specific approaches require the use of mangled names with, which practically forces programmers to limit themselves to C-style interfaces.)

2.3 Unloading

Many useful plugin applications do not require that the plugin be detached from the program prior to termination. However, there are also interesting applications where the ability to unload plugins is essential (e.g., to limit the maximum memory footprint of the application).

Unloading a plugin is unfortunately not trivial. Two problems are the responsibility of the implementation:

- if a plugin has been loaded multiple times (e.g., by two other plugins), it should not be unloaded until all its loaders have requested it to be unloaded (reference counting is often a good solution for this).
- the destruction of namespace scope variables (and static data members) must occur prior to unloading, which may not correspond to a strict "reverse of the order of construction" ordering.

Additionally, the programmer must be careful not to access entities in the plugin after it has been unloaded. Graceful recovery from such attempts would be a plus.

2.4 Plugin discovery

Many applications allow a user to place plugin files in certain conventional locations (file system directories) and the application automatically finds them and loads them if needed. This proposal does not attempt to provide a standard mechanism to provide a matching facility. (However, it is expected that the File System Library proposal for TR2 will provide the building blocks to achieve this.)

2.5 Plugin interface discovery

It is possible to develop a protocol to dynamically discover the interfaces offered by a plugin. A complete C++ solution would also involve the dynamic description of the types involved (aka. run-time reflection). This is desirable territory, but in the interest of simplicity, this proposal does not attempt to address that problem. However, the proposal is not expected to "stand in the way" of future extensions in that direction.

3 A module-based plugin approach

This section proposes a sketch of a module-based approach to supporting plugins in the language.

The proposal assumes a model where plugins are identified by file names. To avoid introducing the notion of files in the core language, the loading/attaching of plugins is cast in terms of standard library facilities. For reasons of symmetry, unloading/detaching a plugin is also expressed through a library function. Entity access is expressed in terms of a core language primitive that maps a loaded plugin onto a module interface. Making this a core language mechanism allows it to deal transparently with various issues that are typically closely tied to the compiler (name mangling, metadata structures like virtual function tables and **`type_info`** variables, etc.). The "glue" between the compiler and the library is a new standard type (this is not unlike **`std::type_info`**).

3.1 Run-time entities

A **run-time entity** is any of the following kinds of program entities:

- a namespace scope variable or function
- a member function or a static data member, including such entities resulting from the instantiation or specialization of a template

(In usual C++ language implementations, these are the entities that are identified using mangled names in object files.)

3.2 std::plugin

A new standard type, tentatively spelled **`std::plugin`**, is proposed. It is exposed to client code as an incomplete class type. I.e., as if declared with:

```
namespace std { class plugin; }
```

3.3 Defining plugins

A plugin is defined as a module whose name is followed by parentheses:

```
export P() {  // Parentheses indicate plugin nature of
public:       // the module.
  struct S {
    void f() {}
  };
  S* g();
  // ...
}
```

Such a module is called a **plugin module**. The fact that a module is a plugin module can cause a compiler to:

- change how namespace-scope variables are destroyed
- provide under-the-cover machinery to support dynamic importing (see below)
- potentially warn that inline functions will not get inlined outside the plugin boundaries

Function templates and member functions/static data members of class templates cannot be instantiated outside a plugin in which they are defined (i.e., there is no requirement to perform run-time template instantiation).

The use of parentheses as an indication of the plugin nature of the module is only tenuous in this context (it is reminiscent of a function, which like a plugin is an entity with run-time effects), but it will become clearer when looking at additional supporting language constructs below. An attribute-based syntax was also considered, but early feedback was negative:

```
[[plugin]] export P {
  // ...
}
```

3.4 Loading plugins

A new library function (or set of overloaded functions?), tentatively spelled **`std::attach`**, is proposed. It takes a file name and returns a pointer to a **`std::plugin`**.

```
std::plugin *p = std::attach("/App/Plugins/p.dyn");
```

If the call is successful, the namespace scope variables and the static data members of the associated module are initialized and a unique pointer identifying the loaded plugin is returned.

The call may fail with a thrown exception for a variety of reasons:

- plugins are not supported (e.g., because of platform limitations or because the program was compiled/linked with options that don't allow for dynamic loading).
- the indicated argument does not correspond to a file.
- the indicated argument does not correspond to an actual plugin.
- the initialization of the module tries to (directly or indirectly) attach itself.
- the initialization of the module failed with an exception.
- some other reason (likely a system limitation of some sort).

3.5 Import expressions

An **import expression** associates a module with a dynamically attached plugin. The syntax is as follows[1]:

```
import N(p)
```

or

```
::import N(p)
```

where **N** is a previously imported plugin module name and **p** is a pointer returned by **std::attach**. Import expressions are void expressions. The first variant is a **local import expression** and has an effect only within the current block scope, or within the current full expression if it appears outside of any block scope. The second variant is a **global import expression** and affects all subsequent accesses to plugin entities not affected by a local import expression. Accessing a run-time entity from a plugin module when no import expression for that module is in effect should either be undefined behavior (easier to specify/implement) or require an exception to be thrown. For example (assuming the plugin definition above):

```
import std;
import P();
S *s;  // Okay: No access to run-time entity of P.
int main() {
  std::plugin *p = std::attach("/App/Plugins/p.dyn");
  { import P(p);  // Local import expression.
    g()->f();     // Okay.
  }               // Local import effect ends.
#ifdef NEG
  s = g();  // Would be undefined or exception.
#endif
  { ::import P(p);  // Global import expression.
  }                 // Effect continues globally.
  g()->f();  // Okay.
}
```

The statements sequence

[1] It is hoped that this justifies the plugin definition syntax.

```
import P(p);
g()->f();
```

could be combined into a single comma-expression:

```
import P(p), g()->f();
```

The fact that module mapping is formulated in terms of expressions (rather than statements) is also useful to allow the use of plugin facilities in initializers for namespace scope variables and data members (both static and nonstatic).

It is expected that local import expressions can be implemented more efficiently in multithreaded environments and that they are more likely to be transparent to optimizers. However, for many applications the persistent effect of a global import expression is what is required.

3.6 Unloading plugins

A new library function, tentatively spelled **`std::detach`**, is proposed. It takes a pointer to a **`std::plugin`** and does not return a value (**`void`** return type).

```
std::detach(p);
```

The argument must be a pointer value returned by **`std::attach`**. (If necessary, null pointers could be defined to have no effect.) The call causes the destructors of namespace scope variables and static data members from the associated module to be run in their reverse order of construction (entities in other modules are not yet destroyed, however). After a plugin has been detached, the effect of an import expression for the associated module is undone: Access to run-time entities of the module will again result in undefined behavior or exceptions.

If a plugin module is not explicitly detached, it could be mandated that it be detached at program termination time or some other requirement could be made. A call to **`std::detach`** could also be made responsible to apply recursively to any plugins still attached to the plugin being detached.

3.7 Mapping to multiple plugins

Applications often load multiple plugins that conform to a single plugin interface. With the proposed language support, this could be exercised using the following code fragment:

```
std::plugin *p1 = std::attach("/App/Plugins/p1.dyn");
std::plugin *p2 = std::attach("/App/Plugins/p2.dyn");
for (int k = 0; k < 10; ++k) {
  import P(p1), g()->f();
  import P(p2), g()->f();
}
```

A practical problem, however, is that care must be taken that the various plugins implementing a certain plugin module are compatible with the declarations in the corresponding module file (Section 5, "Implementation notes", clarifies why this might be an issue). Leaving this as a requirement for the programmer (to carefully declare public entities in the same order in each plugin), seems inadequate. Instead, we propose the possibility of defining a **plugin interface** that is separate from the various **plugin implementations**. Possible syntax for the interface definition might be[2]:

```
register P() {  // Plugin interface
public:
  struct S {
    virtual void f();
  };
  // ...
}
```

and for a corresponding implementation:

```
export for P() { // Plugin implementation
public:
  struct S {
    void f() {}  // Error: Should be virtual
  };
}
```

When compiling a plugin module implementation, a compiler first loads the corresponding compiled plugin module interface and subsequently enforces compatibility between the two.
A plugin module interface might disallow run-time entity definitions. Alternatively, it could allow definitions and use them as a default implementation if no plugin is loaded. Incidentally, a similar interface enforcement mechanism for nonplugin modules could be a highly welcome design and development tool.

4 A complete example

4.1 File 1: A plugin interface

```
register StringSource() {
public:
namespace StringSource {
  import std;
  std::string next();
}
}
```

[2] The use of the keyword `register` as a verb is admittedly contrived. Nonetheless, it is meant to be reminiscent of the fact that the plugin's run-time interfaces must be registered on the plugin client side.

4.2 File 2: A client program

```
import std;
// Import the plugin interface:
import StringSource();

int main() {
  std::plugin *p;
  std::string p_name;
  try {
    // Repeatedly read the name of plugin to be loaded
    // and output the string it produces:
    while (std::cout << "\n? ", std::cin >> p_name) {
      p = std::attach(p_name);
      import StringSource(p);
      std::cout << StringSource::next();
      std::detach(p);
    }
  } catch (...) {
    std::cout << "Plugin access failed\n";
  }
}
```

4.3 File 3: A first plugin instance

```
export for StringSource() {
public:
namespace StringSource {
  import std;
  std::string next() {
    return std::string("Hello, ");
  }
}
}
```

4.4 File 4: A second plugin instance

```
export for StringSource() {
public:
namespace StringSource {
  import std;
  std::string next() {
    return std::string("World!");
  }
}
}
```

4.5 Sample session

Underlined text is console input:

```
? file3
Hello,
? file4
World!
? file5
Plugin access failed
```

5 Implementation notes

This section sketches how things might work "under the hood". It assumes that plugins are implemented in terms of dynamic libraries (also called DLLs or shared libraries). The approach outlined here is not the only practical one, and various system-specific optimization opportunities are known to exist. The goal is primarily to establish the reasonable feasibility of the proposed facility.

5.1 System-specific facilities

Windows and most variants of Unix support similar facilities when it comes to program-directed loading/access/unloading of dynamic libraries.

5.1.1 Microsoft Windows

Windows provides a function **`LoadLibrary`** that takes a C-string (NTBS) describing a DLL file (or an executable file) and returns a "handle" (an alternative function **`LoadLibraryEx`** offers some additional options). DLLs can provide a special entry point (often referred to as **`DllMain`**) to be notified of special events: (1) DLL is loaded, (2) new thread is created, (3) thread exits cleanly, and (4) library is unloaded or process exits cleanly. (The Microsoft C++ compiler runs constructors and destructors of variables with static lifetime using this mechanism.)
Given a handle and a low-level (i.e., mangled) name of an exported variable or routine, the function **`GetProcAddr`** returns a pointer to the associated entity.
A DLL can be unloaded by passing its associated handle to the function **`FreeLibrary`**.
`LoadLibrary` and **`FreeLibrary`** work using a reference count: If **`LoadLibrary`** is called N times for a given DLL, N invocations of **`FreeLibrary`** are needed to actually unload the DLL.

5.1.2 Unix variants (Linux, MacOS X, Solaris, others)

Many variants of Unix offer functions **`dlopen`**, **`dlsym`**, and **`dlclose`** that are very similar to Microsoft's **`LoadLibrary`**, **`GetProcAddr`**, and **`FreeLibrary`**, respectively. In addition to taking a file name locating the dynamic library to load, **`dlopen`** also takes a second integer argument to pass optional flags that control (among other things) how symbols in the loaded dynamic library are available to code that does not have access to the handle returned by **`dlopen`**. (The flags vary from one Unix-based

platform to another.) As with the corresponding Microsoft functions, **`dlopen`** and **`dlclose`** maintain a reference count for a given dynamic library.
Not all Unix applications can successfully "dlopen" a dynamic library: An application must have been linked in an appropriate way for this to be possible (although that is usually the default linking option).

5.2 Compiling a plugin module

When a dynamic library has been loaded by the application using Microsoft's **`LoadLibrary`** or Unix's **`dlopen`**, access to entities in that library must be indirect through the pointers obtained from **`GetProcAddr`**/**`dlsym`**. Although the plugin module feature proposed here shields the programmer from the associated coding tedium, the underlying mechanism remains indirect access.
When compiling a plugin module, the compiler can create (in the resulting object file) a table of pointers to all the exposed (usually: public) run-time entities and to the metadata associated with public module members (specifically, virtual function tables and **`type_info`** objects). In the module file (the other file likely to be produced by the compilation of a module and read by a C++ front end when importing a module — see N2073) the association of an exposed module member or a metadata item with an index into the table can be described.
On the client side, all accesses to run-time entities of a given plugin module can be code-generated in terms of a pointer (simply called **`pix`** henceforth) to the table of public entity pointers in the plugin (this is somewhat similar to the idea of a virtual function table). The **`pix`** pointer can be a module-wide pointer in code that is not in the scope of a local import directive, or it can be a local pointer otherwise.

5.3 Plugin client code generation

A call to **`std::attach`** could result in a call to **`LoadLibrary`** or **`dlopen`** (with the result stored inside the opaque **`std::plugin`** object whose address is to be returned), followed by a call to **`GetProcAddr`** or **`dlsym`** to find the **`pix`** pointer value for that plugin (which is also stored in the associated **`std::plugin`** object). An import expression **`import N(p)`** then amounts to simply loading the **`pix`** pointer associated with module **`N`** with the value saved in **`*p`** by the call to **`std::attach`**. This approach makes import expressions relatively cheap, so that frequent switching between different instances of a plugin module can be practical. **`std::detach`** need not be more than a call to **`FreeLibrary`**/**`dlcose`**.
A viable option might be to initialize a **`pix`** pointer to a table that causes an exception to be thrown. A call to **`std::detach`** could potentially also check if the plugin being unloaded is the one currently accessed through the **`pix`** pointer, and if so reset the **`pix`** pointer to the table that triggers exceptions.

6 Acknowledgments

Benjamin Kosnik, Loïc Joly, Lally Singh, and Thomas Witt contributed significant ideas and insights.