Tutorial - A first plugin (no C code)

Table of contents

1. Tutorial - A first plugin (no C code)

In the last tutorial we saw how to start up the plugin system and how to run some simple plugins. Those plugins either didn't need any extra information to run or could be parameterised by a few simple variable settings. Some plugins, however, need (or can take) more information than is readily available from setting variables. For these we need to write another plugin which provides that information. We'll do that in the this tutorial.

The way we provide more information to plugins is by extending their *extension-points*. Each is *extension-point* will be passed the XML content of any *extension* elements that extend it. The *extension-point* can make whatever sense of this XML it wishes. Often it will expect the XML of the *extension* to name some symbols in a shared library so that it can run code from that library. Equally often, the XML fully describes how the extension should be made.

For example, there is a plugin which prints messages in response to events generated by the application or by other plugins. The *extension-point*s of this plugin can be given a chunk of XML which describes what to print when the event happens. There is no need to write a shared library to use it. There are other such examples through out the plugin system.

2. Setup

We're going to build a plugin file. To start with we'll make the most basic, "Hello, World!", style of plugin. It will work by extending an extension point of another plugin, asking it to print a string for us.

We'll keep the setup from the last tutorial as a starting point, really just so that it gives us some files to play with. In the directory above *tutorial-1*, type:

```
mkdir tutorial-2
cp tutorial-1/foo.c tutorial-2/
cd tutorial-2
```

We'll also need the plugin that we make in this tutorial to be in the plugin path. Previously, the plugin path was set to *plugin-dir*, the directory where we unpacked the downloaded files. Now, we'll also include the current directory:

```
export GCC_PLUGIN_PATH=.:plugin-dir
```

We could, instead, have directly pointed to the new plugin file. This is useful if there are mal-formed XML documents in the same directory (we'll be creating the file in the next section):

```
export GCC_PLUGIN_PATH=hello-world.xml:plugin-dir
```

3. The "Hello, World!" plugin

So, let's start right off by creating a file called *hello-world.xml* in the current directory, *tutorial-2*. The contents of the file will be as below (except without the line numbers which have just been added for clarity).

```
01 <?xml version="1.0"?>
02
03 <?plugin version="4.3"?>
04
05 <plugin id="hello-world">
06     <extension point="message.start">Hello, World!</extension>
07 </plugin>
```

Run the compiler with that plugin over the *foo.c* file from the last tutorial and then we'll discuss what each of the lines in the file mean.

```
plugin-gcc -o foo foo.c; echo
```

We should see:

```
Hello, World!
```

Printed in the terminal. Success! (The file should be compiled, too).

Warning:

If you didn't copy the files over and just built the plugin file in the old directory, *turorial-1*, you might have got a list of errors complaining about *perfmon.xml* not being valid XML. Now the plugin system searches through any **.xml* files in the plugin path to see if they might be plugin files. If any of them is not well formed it will complain - **Only have well formed XML files in your plugin path**.
If you happened to run the *gcc.perfmon* plugin twice in the last tutorial, the *perfmon.xml* file may not be well formed anymore (having more than one top level element). This will cause the plugin system to fail.
Generally it is probably better to have a separate directory for plugins so that partially complete and broken XML files don't interfere. Alternatively, directly point to plugin files, not directories in the plugin path.

So, what was in the *hello-world.xml* file? Let's look at it line by line.

Line 01 is a standard piece of XML declaring the file to be XML version *1.0*. You don't have to have this line at all, but it is good practice.

Note:

There is also XML version *1.1*. However, the library used to do XML processing by the plugin system is *libxml2* which only likes XML version *1.0*.

Line 03 is an application version processing instruction. This is used by the plugin system to check that file really is a plugin file - all other XML files are ignored. Since there has to be a bone fide application version, one plugin path can hold plugins that work in different applications and there will be no confusion with the plugin system trying to load plugins that don't match the current application. Application version processing instructions also ensure that you may have other types of XML file in the plugin directory (as long as they are well formed).

In the case of GCC there are two types of acceptable processing instruction (either of which must be at the top level). They are:

```
<?plugin version="0.1.0"?>
  -- or --
<?gcc version="any"?>
```

The first should be used when your plugin could be used with any application that uses *libplugin*.

The second should be used when your plugin needs something specific to GCC (either in the compiler itself or in some other plugin it needs). At the moment the content of the GCC version string is ignored (you must have a version string but it doesn't matter what it is). That may change in the future so it is best to ensure that the string matches the version of GCC you expect.

Line 05 is the start of the plugin declaration. Every plugin file has this element as the root element. The *@id* attribute gives the identifier of the plugin which must be unique amongst all plugins.

Note:

At some point in the future, plugins will support different versions being available at the same time. When that happens the requirement that plugin identifiers be unique will be relaxed.

Line 06 is the meat of the plugin file. It states that we wish to extend the extension point called *message.start*. The system will try to find that extension point, and will load the owning plugin if it hasn't already been loaded.

All extension points have a method, *extend*, which takes the XML of from an *extension* element and 'does the extension'. In the case of the *message.start* extension point it will remember any text passed to it and will print it out when the plugin system starts (i.e. has successfully loaded).

It's as simple as that (although the message plugin is really more powerful).

Line 07 closes the plugin element and makes the document well formed.

Congratulations! You've built your first plugin!

4. Another first plugin - changing command line arguments

Well, that last plugin was nice, but not hugely useful. Here we're going to do something that's useful for iterative compilation. Iterative compilation repeated compiles programs with slightly different strategies to find out which one it best. Often this just means selecting different command line options.

Now, it's sometimes quite painful to have to go changing makefiles to do this. Wouldn't it be nice if there was a plugin that could help? Oh, wait, there is!

The plugin is called *command-line*. It provides an extension point called *command-line.modify* which changes the command line arguments. By extending it with our own plugin and then having environment variables cause that plugin to be loaded we could make the command line arguments to GCC change without having to edit the makefile.

Note:

The command-line plugin, when used with GCC doesn't affect the driver program, *gcc*, only the actual compilers, like *cc1*. This means that you can't change all the parameters, only the ones that make it through. To be clear, as you will see shortly, the command line arguments to *cc1* are different to the ones passed to *gcc*.

Well, let's start right up with a plugin. Here's what we'll do: we'll remove any *-O* option and insert *-O3 -funroll-all-loops -fno-inline*. For iterative compilation, of course, we'd need a driver program that continually spewed out different strategies, each encapsulated as a plugin which sets up different command line arguments.

```
<?gcc version="4.3"?>

<plugin id="change-command-line">
        <extension point="command-line.modify">
                <remove><arg>-O*</arg></remove>
<insert><arg>-O3</arg><arg>-funroll-all-loops</arg><arg>-fno-inline</arg></insert>
        </extension>

        <extension point="message.start">
                <text>The command line used is :</text><print
import="print.command-line"/><br/>
        </extension>
</plugin>
```

That's it! You should know how to run this plugin against your code already. See what happens when you do.

You may have noticed that we've used the *message.start* extension point again, but this time with more complex arguments. You can find out about that in the *message* plugin's documentation. For now, though, all you really need to know is that it prints out the command line after it gets modified.

5. Yet another first plugin - changing controlling loop unrolling

We'll just add one more, just for luck! In this one we're going to affect the loop unrolling factor of the loop in our tiny program, *foo.c*. We'll just dive right in and write the plugin, again, you should be able to run it and you'll find more details in the plugin documentation.

```
<?gcc version="4.3"?>

<plugin id="change-command-line">
        <extension point="command-line.modify">
                <remove><arg>-O*</arg></remove>
<insert><arg>-O3</arg><arg>-funroll-all-loops</arg></insert>
        </extension>

        <extension point="gcc.rtl-unroll-and-peel-loops.override">
                <loop main-input-file="foo.c" function="main" number="1"
times="20"/>
        </extension>
</plugin>
```

We had to change some command line arguments to ensure that loop unrolling happened at all. Then we told it to unroll the first loop in *foo.c:main* twenty times.

And if we want to see if it really worked, we can add the following line inside the *plugin* element. It will print all the loops that were unrolled to a file called *unrolled-loops.xml*.

```
        <requires plugin="gcc.print.unrolled-loops"/>
```