Tutorial - Using existing plugins in GCC

Table of contents

1. Tutorial - Using existing plugins in GCC

This tutorial takes you through your first steps using the plugin system in GCC. You'll see how to tell GCC which plugins to load and you'll use some of the existing GCC plugins to play with GCC's behaviour.

Some plugins can be used just by telling the plugin system to load them. Others require more complex parameters and have to be initialised by writing another plugin. In this tutorial you won't build any of your own plugins, you'll just see how to run some interesting plugins that work by themselves (perhaps with a couple of variables set). In a later tutorial you'll build your own plugins to do more refined things.

In this tutorial you will learn to:

- Turn logging on and off in the plugin system
- Set the plugin path
- Choose specific plugins to load
- Set plugin variables
- Print a list of the passes that are run while compiling your code
- Instrument code to collect cycle count information

2. Setup

First of all you need to go through the install process, of course, [click here]. And, of course, you need to do all the GCC bits of that, too. We'll assume that you have the newly built gcc in your path as *plugin-gcc* and that the rest of the plugin system is installed at *plugin-dir*.

Since we're going to be controlling GCC which compiles things, we'd better have something to compile. So, let's start by making a directory to build some code in:

```
mkdir tutorial-1
cd tutorial-1
```

Now we'll put a file in there called *foo.c*. The file will look like:

```
#include <stdio.h>

int main( int argc, char* argv[] ) {
        int i;
        for( i = 0; i < argc; ++i ) {
                printf( "%d ", i );
        }
        printf( "\n" );
}
```

If everything has gone as planned, you should be able to compile and run it:

```
plugin-gcc -o foo foo.c
./foo
```

3. Logging

The plugin system can produce verbose logging information. This tells you what plugins have been loaded, what extensions are available, etc. This is often useful for trouble shooting why your plugin hasn't loaded or some extension point isn't found. In gcc, logging can be turned on by setting the *GCC_PLUGIN_LOGGING* environment variable to anything.

So, if we type the following:

```
GCC_PLUGIN_LOGGING=true plugin-gcc -o foo foo.c
```

We'll see a few lines of logging, something like:

```
<path-to-cc1> - Log:Plugin system starting with path: (null)
<path-to-cc1> - Log:Plugin system started
<path-to-cc1> - Log:Plugin system stopping
<path-to-cc1> - Log:Plugin system stopped
<path-to-cc1> - Log:Plugin system cleaning up
<path-to-cc1> - Log:Plugin system cleaned up
```

Now, when the plugin path contains some plugins there'll be a huge amount of logging, so from now on we won't use it very often.

4. Setting the plugin path

The plugin path lets GCC know where it can find plugins, without setting it, no plugins can ever be loaded. There are a couple of ways to set the plugin path.

The first is by setting an environment variable, *GCC_PLUGIN_PATH*. This path should be a list, separated by ':' characters, just like the system path.

The second is by giving a command line argument, *-plugin-path <path>*. Again this is a colon separated list. This list will be appended to, not replace, the path from the environment variable, *GCC_PLUGIN_PATH*.

The environment variable route is great for those times when you want to use plugins on an existing make file which you don't want to alter. It also lets you set the default plugin to be

always used.

There are a number of plugins in the distribution, so we can add that directory to the plugin path to make the plugins there available. So, we type:

```
export GCC_PLUGIN_PATH=plugin-dir
```

If you try compiling the file, *foo.c* again, with logging on, you should now get lots of output.

```
GCC_PLUGIN_LOGGING=true plugin-gcc -o foo foo.c
```

We could have got the same effect from:

```
GCC_PLUGIN_LOGGING=true plugin-gcc -plugin-path plugin-dir -o foo foo.c
```

The elements of the plugin path can directly point to plugin files. They may also point to directories. Any XML file in such a directory is a candidate for a plugin file - therefore they should be well formed XML or the plugin system will fail. Directories may also contain sub directories. If any of those contains a *plugin.xml* file, that will be a candidate plugin file.

5. Choosing which plugins to load

So far, we haven't actually changed GCC in any way. No plugins have actually been loaded. Part of the reason for this is that all the plugins in the distribution are marked as *lazy*. That means they aren't loaded unless you specifically ask for them or some other plugin you've asked for needs them. You can make your own plugins which are always loaded, but we'll leave that for later.

Each plugin has an identifier, which can be found from the *@id* attribute in its plugin file (usually the same or similar to its directory). These ids are used to specify which plugins to load.

Again, there are two ways to specify which plugins should be loaded. One is through an environment variable and the other via the command line. The two mechanisms are additive, so you can use the environment variable to always load you preferred plugins and the command line arguments for per file plugins.

The environment variable is *GCC_PLUGINS* and the command line argument is *-plugins*. Both take a comma separated list of plugin specifiers. The basic specifier is just the id of the plugin you want to load, but they can also be glob type patterns.

Suppose we want to find out what passes are run when we compile *foo.c*. Fortunately, there

is a plugin, already built, which performs just this task. It's id is *gcc.print.passes* and it is found in the distribution in file *gcc.print.passes.xml*. We can load it with:

```
plugin-gcc -plugins gcc.print.passes -o foo foo.c
```

You will now notice a new file in the tutorial directory, called *passes.xml*. It contains something like:

```
<passes compilation-unit="foo.c">
    <pass id="remove-useless-stmts" name="useless" function="main"/>
    <pass id="mudflap-1" name="mudflap1" function="main"/>
    <pass id="lower-omp" name="omplower" function="main"/>
    <pass id="lower-cf" name="lower" function="main"/>
    <pass id="refactor-eh" name="ehopt" function="main"/>
        ...
    <pass id="final" name="<null>" function="main"/>
    <pass id="df-finish" name="dfinish" function="main"/>
    <pass id="clean-state" name="<null>" function="main"/>
</passes>
```

You can specify multiple plugins at once. Try this (and then see what files are present in the tutorial directory):

```
plugin-gcc -plugins 'gcc.print.passes, gcc.*loops' -o foo foo.c
```

Note:

The file, *passes.xml*, has been appended to, not replaced. This is important because often you are compiling multiple files at once. GCC actually calls *cc1* once for each file so if the plugin replaced the file, you would only get passes for the last file.

Note:

The two plugins that generate the files, *unrolled-loops.xml* and *unrollable-loops.xml*, are *gcc.print.unrolled-loops* and *gcc.print.unrollable-loops*. You will notice that they don't produce much interesting output. This is because the pass they need to be executed is not run by default (the pass is *pass_rtl_unroll_and_peel_loops* from *loop-unroll.c*.
If you instead type:

```
  plugin-gcc -plugins 'gcc.print.passes, gcc.*loops' -O3 -funroll-all-loops -o foo
foo.c
```

you'll get more interesting results.
You can actually alter the command line arguments via another plugin (whose id is *command-line*). But for that you need to write another plugin to tell it how to alter the command line. This is something we'll come to in another tutorial.

6. Setting plugin variables

Plugins can accept variables from the command line or from environment variables so that they can change their behaviour. Here we'll find out how to pass variables to plugins.

The plugin, *gcc.print.passes*, normally prints it's output to a file called *passes.xml* in the current directory. If you look inside the plugin file itself, *gcc.print.passes.xml*, you will see the following line:

```
<variable name="${plugin.id}.file" value="passes.xml" overwrite="false"/>
```

This sets the default value of the variable, *gcc.print.passes.file* to *passes.xml*. That variable is then used to determine where to write the output. You can find more details about the plugin file format if you [click here].

Note:

You'll see that the definition of the variable name uses another variable, *plugin.id*. This is defined by the plugin system and in this case evaluates to *gcc.print.passes*.

We can change the variable value by using the command line:

```
plugin-gcc -plugins gcc.print.passes -plugin-var
gcc.print.passes.file=/dev/stderr -o foo foo.c
```

Now, the pass list will be printed to the standard error stream.

You can specify multiple variables, separating them by ':' characters. You can escape colons and '=' with '%' characters (and you can escape '%' by doubling up, '%%').

There's also an environment variable, *GCC_PLUGIN_VAR*, which sets variables in the same way. The command line variables add additional variables to those set in the environment variable and override any with the same name.

Note:

You shouldn't have to look inside plugins to find out what variables it supports. The plugin's documentation should provide that information (as is the case for plugin *gcc.print.passes*).

7. Instrumenting for cycle counts

So far the plugins we've used have only reported what has been happening in the compiler. Now we'll look at a plugin which does something more interesting. This plugin, *gcc.perfmon*, lets you gather cycle counts and call count information about your program. It isn't meant to replace a professional performance monitoring system, but it can be useful and demonstrates some principles. In particular it can be useful because there is no need to link against some performance monitoring library. As long as you link against the C library, it should be fine.

The plugin provides a number of customisation options, but for the moment we'll use it in it's

simplest form.

All we have to do is to type:

```
plugin-gcc -plugins gcc.perfmon -o foo foo.c
./foo
more perfmon.xml
```

We've loaded the *gcc.perfmon* plugin. This instruments each function in every file you compile with it. The functions get code added to the beginning and end to count the number of calls and to count the number of cycles used in the function. The cycle counter is paused every time one function calls another so that the cycle counts for a callee don't contribute to the count of the caller.

Finally, in each compilation unit a function to print all the statistics is inserted to run when the program finishes. By default these data are written out to *perfmon.xml*.

You'll find the output looks something like this:

```
<perfmon main-input-file="foo.c">
        <function name="main"
                call-count="1"
                cycle-count="286"
                pause-count="2"
                total-cycle-count="85813"/>
</perfmon>
```

Which shows the cycle counts for functions in translation unit *foo.c* (if you had more than one translation unit, you'd get more top level elements). You see how many times the function was called and how many cycles were spent in the function (not including other function calls - which is why the *total-cycle-count* is so large, it includes the printfs). The *pause-count* tells you how many times the cycle count for function was paused so that another function could be called.

More information can be found in the documentation for the *gcc.perfmon* plugin.

Warning:

At the moment the *gcc.perfmon* plugin is not cross platform. It only works with platforms where the *RTDSC* instruction is present - which practically limits you to modern x86 machines.

8. Conclusion

You've seen how to start up the plugin system and how to run some simple plugins. There are more plugins that can be run in a similarly simple fashion (see the documentation for

individual plugins).

Some plugins, however, need more information than is readily available from setting variables. For these you need to write another plugin which provides that information. You'll do that in the next tutorial, [click here].