Uninstalled pkg-config files

Emilua 0.5 has shipped with support for a little known feature from pkg-config: `*-uninstalled.pc` files. This article explores how they improve the productivity for plugin developers.

Emilua plugins

Emilua's primarily targets Lua programmers. However it's impossible to do anything useful in a PC completely ignoring C, so it has support to load plugins written in C++ when one must.

Deploying Emilua modules can be summarized in three steps:

1. Build.
2. Ship.
3. Load.

For Lua modules, "build" equals to copy some directory over to a destination. For C++ plugins, this task is much more complex (just like anything is more complex in C++). The rest of the steps are the same whether you're coding in Lua or C++. The rest of the text will focus on C++ specifics (build).

Building a C++ project

C++'s ecosystem is a zoo of non-coordinated efforts struggling to work together after enough patching and hammering is performed.

For POSIX, pkg-config became long ago the de facto standard to propagate dependency properties up in the stack. Windows didn't have a real standard until CMake grabbed the huge market share it enjoys today. Some CMake projects are kind enough to also generate pkg-config for non-Windows projects, but that's not very relevant in Windows itself. Some build systems such as Meson are smart enough to look for installed dependencies through either pkg-config or CMake, and play very well on both Windows and non-Windows systems.

However that only deals with a metadata standard to propagate build flags you pass up to the compiler. This step only deals with *dependencies that are already installed*. One still needs to do actual dependency management. We'll leave dependency management aside and focus on the layer pkg-config.

Emilua gives you all the variables your project needs to build and install on the running platform through pkg-config. However when one is developing multiple layers of plugins (e.g. the plugins GLib and GI choreograph together to offer bindings for the GTK+ ecosystem), it's useful to be able to quickly iterate changes over any plugin in the stack and bypass installation steps altogether. GStreamer developers faced a similar problem and give support for the same approach that Emilua recently adopted to solve this problem.

Uninstalled *.pc files

The solution pkg-config offers is to give preference for any `pkg-config` file whose stem ends in `-uninstalled` if one is found. Then your role is to generate two pkg-config files. One will be just the usual. Meanwhile the `*-uninstalled.pc` one will actually refer to paths in the build directory. That's the trick.

Meson has had some support for uninstalled pkg-config files since v0.54.0. Emilua has made use of this support since v0.5.0. `emilua-uninstalled.pc` will be generated in `$BUILD_DIR/meson-uninstalled`. Just add `$BUILD_DIR/meson-uninstalled` to `$PKG_CONFIG_PATH` and your build system will be able to detect Emilua from its build directory. The same setup can be performed for plugins.

Guidelines for writing uninstalled pkg-config files

I can't really show what to write in your uninstalled pkg-config files as that will vary from C++ project to C++ project. I could write a section on guidelines, but then again, this section would only have one guideline:

- Uninstalled pkg-config files must refer to paths from your build directory.

So the best I can do is to show how I use this feature today. Maybe an example is the best teacher you can have on this topic.

Doing it step-by-step

The process is not complex at all, but it's showcased here anyway just in case.

First clone and build Emilua as usual:

```
git clone "https://gitlab.com/emilua/emilua.git"
pushd emilua
meson setup build
meson compile -C build
export PKG_CONFIG_PATH="$PKG_CONFIG_PATH:$(realpath build/meson-uninstalled)"
popd
```

You don't need to build Emilua itself in this way if you already have it installed in the system and no changes are planned for Emilua codebase itself. However it still might be useful to build Emilua in this way so plugins can be installed without affecting the rest of the system (details below).

Then clone and build some plugin:

```
git clone "https://gitlab.com/emilua/glib.git" emilua-glib
pushd emilua-glib
meson setup build
meson compile -C build
```

```
export PKG_CONFIG_PATH="$PKG_CONFIG_PATH:$(realpath build/meson-uninstalled)"
popd
```

Do notice how these steps are almost an exact copy of the same commands as before.

Now you can even install the final plugin in emilua's build dir if you feel like it:

```
meson install -C emilua-glib/build --tags runtime
```

You can keep cascading builds in this fashion for as long as it's needed to build the whole chain of plugins you stacked together. Let's do it one last time:

```
git clone "https://gitlab.com/emilua/gi.git" emilua-gi
pushd emilua-gi
meson setup build
meson compile -C build
meson install -C build --tags runtime
export PKG_CONFIG_PATH="$PKG_CONFIG_PATH:$(realpath build/meson-uninstalled)"
popd
```

Then when it's time to actually start emilua and test your plugins, just remember to add each path individually in `EMILUA_PATH` or — if you've run `meson-install` to install every plugin in emilua's build dir — just set `EMILUA_PATH` to the single directory `emilua/build/plugins`.

```
export EMILUA_PATH="$(pkg-config --variable=pluginsdir emilua)"
"$(pkg-config --variable=emilua_bin emilua)" your_test_program.lua
```

Why is it faster to iterate changes in this way?

Well just you try to make changes in any files and see for yourself. When you trigger a rebuild from projects down the chain they'll only rebuild what's needed. Incremental compilation will play its role and you also skip the installation step altogether.

It's also specially time saving on POSIX systems where you'll no longer need to fight directory permissions to meddle with system directories. Even kernel developers are supporting this type of workflow nowadays to improve productivity.

Maybe you don't find the savings that much important for a project whose build isn't that slow such as Emilua, but consider GStreamer. My current machine has GStreamer and 185 plugins installed. Now remember that GStreamer codebase deals with complex algorithms — audio and video codecs — and this code might not compile as smoothly as one would do with Emilua.

An environment manager for Emilua

It wasn't the intention, but now it became trivial to develop a version manager for Emilua if the need ever arises in the future. Python already has pyenv. Golang has gvm. NodeJS has nvm. Ruby has RVM. Is it easy to develop something like them for Emilua? Now it is. Do we need it? Not yet.

Meson's wrap system

Complementary to uninstalled pkg-config files we have Meson's wrap system. If one is building more and more code relying ever less and less on the system environment then it's also useful to have a package manager that works at a project level. Meson's wrap system is just one of these options and Stephen Brennan has a good introduction on it.

It's completely possible to create an environment manager for Emilua without the need to use uninstalled pkg-config files. One just has to be careful about paths configured during the project's build and override runtime env vars properly so the system's Emilua is never picked up by the environment manager.

Pkg-config

A lot was said about Emilua in this article. However the main point here was on how to make use of a feature from pkg-config that not many people know about. If it makes sense for your project as well, you should already understand the expected workflow by now.