A glimpse of code development using Spack

Kyle J. Knoepfel

23 January 2024

LArSoft coordination meeting

Code development using Spack

This has meant multiple things over the years:

1. FNAL-created `spack dev` *(a replacement for MRB)*

   LArSoft minimum viable product released in 2019; no response from experiments.

2. Upstream-provided `spack develop`

   It works, but there's a high barrier for entry for users.

3. Just providing externals required for development via `spack load`/`env`

   The work presented here.

Chosen approach

The goal is to develop multiple packages using Spack to provide external libraries

Try to give a familiar feel to MRB, but retain only those things most commonly used

Chosen approach

The goal is to develop multiple packages using Spack to provide external libraries

Try to give a familiar feel to MRB, but retain only those things most commonly used

For now:

- `mrb newDev (n)`
- `mrb gitCheckout (g)`
- `mrb install (i)`
- `mrb test (t)`
- `mrb zapBuild (z)`
- `mrb zapDist (zd)`

Let me know if there are any you really think you need.

I already know about `mrbsetenv`, `mrbslp`, `mrb uc` *and* `mrb uv`.

Lessons learned

- **Each repository you want to develop should have a Spack recipe**

  This is not strictly true, but if you don't do this, (a) it'll likely be harder to setup external libraries, and (b) you won't be able to use that repository as a Spack package.

  The recipe need not be part of the Spack mainline repository (e.g. `fnal_art`). Chaining multiple Spack package repositories together is a scalable way to distribute packages.

  The recipe can be very simple (spack can create a skeleton recipe for you)

Lessons learned

- **Each repository you want to develop should have a Spack recipe**

  This is not strictly true, but if you don't do this, (a) it'll likely be harder to setup external libraries, and (b) you won't be able to use that repository as a Spack package.

  The recipe need not be part of the Spack mainline repository (e.g. `fnal_art`). Chaining multiple Spack package repositories together is a scalable way to distribute packages.

  The recipe can be very simple (spack can create a skeleton recipe for you)

- **You should not rely on the presence of specific environment variables**

  Spack recipes can (and do) set environment variables during (e.g.) `spack load`. But when developing that code (i.e. building it) outside of Spack, those variables will either need to be set explicitly or the code adjusted.

  Resulted in various PRs to Cetmodules (thanks, Chris!) and changes to the art-suite packages to significantly reduce dependence on environment variables.

Demo

- The following assumes:
  1. You already have Spack set up (instructions not given today)
  2. You have a local Spack instance where you can install stuff (instructions not given today)
  3. The packages under development are CMake-based

- This demo may help answer some questions but certainly not all.

  I will answer what I can and delegate other questions to more expert individuals.

- We know that documentation is important and needs to be fleshed out.

Backup

Spack MRB

Generate Spack MRB area using:

```
spack mrb \
   --name art-devel \
   --top ${PWD} \
   -D srcs/ %gcc@13.2.0 cxxstd=20
```

```
$ tree -L 1 .
.
├── build
├── local  ← Contains local Spack repository
└── srcs
```

Spack MRB

Generate Spack MRB area using:

```
spack mrb \
   --name art-devel \
   --top ${PWD} \
   -D srcs/ %gcc@13.2.0 cxxstd=20
```

```
$ tree -L 1 .
.
├── build
├── local  ← Contains local Spack repository
└── srcs
```

```
$ tree -L 2
.
├── build
│   ...
├── local
│   ├── install
│   ├── packages
│   ├── repo.yaml
│   └── setup.sh
└── srcs
    ├── art
    ├── art-root-io
    ├── canvas
    ├── canvas-root-io
    ├── cetlib
    ├── cetlib-except
    ├── CMakeLists.txt
    ├── CMakePresets.json
    ├── critic
    ├── fhicl-cpp
    ├── gallery
    ├── hep-concurrency
    └── messagefacility
```