Simple cleanups to LArSoft

Kyle J. Knoepfel
26 March 2019

Motivation

For LArSoft, I often see developers adding code, but rarely see developers removing it. Reasons for this:

- *Maybe you are concerned about breaking downstream code*
- *Maybe you don't have the time*
- *You might need it later*
- *Maybe you don't care*
- *Maybe you don't know that you should care*

Motivation

For LArSoft, I often see developers adding code, but rarely see developers removing it. Reasons for this:

- *Maybe you are concerned about breaking downstream code*
- *Maybe you don’t have the time*
- *You might need it later*
- *Maybe you don’t care*
- *Maybe you don’t know that you should care*

Why should you care?

Motivation

As software projects evolve, they often get larger. This isn't a bad thing, *per se*, but it has consequences:

- The code takes longer to build
- The installed software takes up more space
- The code becomes harder to keep working
- The code becomes harder to understand

Motivation

As software projects evolve, they often get larger. This isn't a bad thing, *per se*, but it has consequences:

- The code takes longer to build
- The installed software takes up more space
- The code becomes harder to keep working
- The code becomes harder to understand

Unless developers proactively take steps to keep things maintainable, the code base will continue to grow until it becomes too wieldy.

Motivation

As software projects evolve, they often get larger. This isn’t a bad thing, *per se*, but it has consequences:

- The code takes longer to build
- The installed software takes up more space
- The code becomes harder to keep working
- The code becomes harder to understand

Unless developers proactively take steps to keep things maintainable, the code base will continue to grow until it becomes too wieldy.

Today, I want to discuss simple ways of cleaning up LArSoft code. Specifically, **the changes suggested today do not relate to software design**. They are guidelines that can be adopted as you go.

Setting the stage

According to running `cloc` over the `develop` branch of the LArSoft packages, LArSoft has anywhere from 300-500K lines of code:

```
---------------------------------------------------
Language          files    blank   comment     code
---------------------------------------------------
C++                1177    75446     67111   257680
C/C++ Header       1010    31044     67529    67473
CMake               260     1226       859     6438
XML                  20      195       294     5221
---------------------------------------------------
SUM:               2467   107911    135793   336812
---------------------------------------------------
```

Setting the stage

According to running `cloc` over the `develop` branch of the LArSoft packages, LArSoft has anywhere from 300-500K lines of code:

```
--------------------------------------------------------
Language           files      blank    comment      code
--------------------------------------------------------
C++                 1177      75446      67111    257680
C/C++ Header        1010      31044      67529     67473
CMake                260       1226        859      6438
XML                   20        195        294      5221
--------------------------------------------------------
SUM:                2467     107911     135793    336812
--------------------------------------------------------
```

Wilson Hall 9th floor guidance:

- Strive to make commits that remove more code than they add.
- *The easiest code to maintain is the code that doesn't exist.*

Step 1: Remove unnecessary files

Remove files *that you know* are not needed. This may take approval from the collaboration.

Examples of this include:

- Code that is not built/installed.
- Empty files (or those only with comments)
- Any *art* module separated into a header and `.cc` file (only `.cc` needed)

Step 2: Remove unnecessary header dependencies

I did a test to see how much time it takes to build `SimWire_module.cc`. I then systematically removed code to gauge the effect of the headers vs. the code in the file.

Step 2: Remove unnecessary header dependencies

I did a test to see how much time it takes to build `SimWire_module.cc`. I then systematically removed code to gauge the effect of the headers vs. the code in the file.

Built code	Build time[1]
Entire file	11.3 s
Only headers	8.0 s
Only *art* headers	5.0 s
Empty file	0.4 s

1 The build time includes the overhead of running `ninja`, as well as preprocessing, compiling, and linking.

Step 2: Remove unnecessary header dependencies

I did a test to see how much time it takes to build `SimWire_module.cc`. I then systematically removed code to gauge the effect of the headers vs. the code in the file.

Built code	Build time[1]
Entire file	11.3 s
Only headers	8.0 s
Only *art* headers	5.0 s
Empty file	0.4 s

1 The build time includes the overhead of running `ninja`, as well as preprocessing, compiling, and linking.

Due to header guards, it's difficult to know who contributes the most. Bottomline, **remove unnecessary headers**.

Step 2: Remove unnecessary header dependencies

But what's an unnecessary header?

- Straightforward to `.cc` files. But if someone is relying on a header dependency in a header file, then removing an "unused" header can break downstream code.

Step 2: Remove unnecessary header dependencies

But what's an unnecessary header?

- Straightforward to `.cc` files. But if someone is relying on a header dependency in a header file, then removing an "unused" header can break downstream code.

Proposal: **LArSoft should adopt a policy where header files include the minimum number of header dependencies.**

Step 2: Remove unnecessary header dependencies

But what's an unnecessary header?

- Straightforward to `.cc` files. But if someone is relying on a header dependency in a header file, then removing an "unused" header can break downstream code.

Proposal: **LArSoft should adopt a policy where header files include the minimum number of header dependencies.**

Discouraged

```
// MyService.h
//   The following headers are used
#include <vector>

//   The following headers are not used
#include "mf/.../MessageLogger.h"
#include "art/.../ServiceHandle.h"
```

Encouraged

```
// MyService.h
//   The following headers are used
#include <vector>
```

Step 2: Remove unnecessary header dependencies

But what's an unnecessary header?

- Straightforward to `.cc` files. But if someone is relying on a header dependency in a header file, then removing an "unused" header can break downstream code.

Proposal: **LArSoft should adopt a policy where header files include the minimum number of header dependencies.**

Discouraged

```
// MyService.h
//   The following headers are used
#include <vector>

//   The following headers are not used
#include "mf/.../MessageLogger.h"
#include "art/.../ServiceHandle.h"
```

Encouraged

```
// MyService.h
//   The following headers are used
#include <vector>
```

- This may break downstream code, but the fix is usually straightforward, and the savings is worth it.

Step 3: Remove unnecessary link-time dependencies

The `SimWire` test from earlier:

Built code	Build time
Entire file	11.3 s
Only headers	8.0 s
Only *art* headers	5.0 s
Empty file	0.4 s

All steps included linking time. If we reduce the number of linked libraries…

Step 3: Remove unnecessary link-time dependencies

The `SimWire` test from earlier:

Built code	Build time
Entire file	11.3 s
Only headers	8.0 s
Only *art* headers	5.0 s
Empty file	0.4 s
Empty file + only art libraries	**0.3 s**

Reducing number of linked libraries generally results in minor savings in build time. The benefits are seen elsewhere (library sizes, run-time overhead, maintenance).

Step 4: Remove unnecessary functions

A common pattern:

```cpp
class MyProducer : public art::EDProducer {
public:
  MyProducer(fhicl::ParameterSet const&);
  ~MyProducer();

private:
  void produce(art::Event&) override;
  void beginJob() override;
  void endJob() override;
};
```

Step 4: Remove unnecessary functions

A common pattern:

```
class MyProducer : public art::EDProducer {
public:
  MyProducer(fhicl::ParameterSet const&);
  ~MyProducer();

private:
  void produce(art::Event&) override;
  void beginJob() override;
  void endJob() override;
};
```

And then later on:

```
MyProducer::~MyProducer() {}
void MyProducer::beginJob() {}
void MyProducer::endJob() {}
```

Step 4: Remove unnecessary functions

If there is no work to be done at `{begin,end}{Job,Run,SubRun}` for producers, filters, or analzers, do not provide an override:

```
class MyProducer : public art::EDProducer {
public:
  MyProducer(fhicl::ParameterSet const&);

private:
  void produce(art::Event&) override;
};
```

Step 5: Remove inappropriate preprocessor use

There some places where preprocessor macros are being used when they shouldn't be:

- ROOT no longer supports the `__GCCXML__` preprocessor variable. It has been replaced by `__ROOTCLING__`.
- Do not place header guards in implementation files.
- Do not `#define PI 3.1415`

Step 6: Simplify the code (1)

Defining *art* modules:

```
- namespace something {
-   DEFINE_ART_MODULE(MyModule)
- }
+ DEFINE_ART_MODULE(something::MyModule)
```

Step 6: Simplify the code (1)

Defining *art* modules:

```
- namespace something {
-   DEFINE_ART_MODULE(MyModule)
- }
+ DEFINE_ART_MODULE(something::MyModule)
```

Iterating over `std::map` entries:

```
- for (auto const& pr : some_map) {
-   auto const& key = pr.first;
-   auto const& value = pr.second;
-   ...
- }
+ for (auto const& [key, value] : some_map) {
+   ...
+ }
```

Step 6: Simplify the code (2)

Creating `std::unique_ptr`s:

```
- std::unique_ptr<MyType> p(new MyType(arg1, arg2, ...));
- auto p = std::unique_ptr<MyType>(new MyType(arg1, arg2, ...));
+ auto p = std::make_unique<MyType>(arg1, arg2, ...);
```

Step 6: Simplify the code (2)

Creating `std::unique_ptr`s:

```
- std::unique_ptr<MyType> p(new MyType(arg1, arg2, ...));
- auto p = std::unique_ptr<MyType>(new MyType(arg1, arg2, ...));
+ auto p = std::make_unique<MyType>(arg1, arg2, ...);
```

Nested namespaces:

```
- namespace a {
-   namespace b {
-     ...
-   }
- }
+ namespace a::b {
+   ...
+ }
```

Feature branches

I am working on some `feature/knoepfel_cleanups` branches for LArSoft.

The feature branches include:

- Removal of `__GCCXML__` preprocessor directives
- Removal of header guards from module implementation files
- Removal of some unnecessary functions
- Removal of some unnecessary header dependencies
- Removal of many unnecessary link-time dependencies

This changes have removed a few thousand lines of code. Many of the changes have been committed to LArSoft's develop branches, but there are more to go (and I have to make sure I don't break downstream code).

Next steps

Will give Lynn a concrete list of feature branches in the next week or two.

I think LArSoft would benefit from developing several policies:

- When should header dependencies be introduced?
- When should link-time dependencies be introduced?
- What should the header-guard convention be?
  - *art* has an automated header-guard generator.
- What about error handling (*mea culpa*)?
  - I'm seeing a lot of `cet_enable_asserts()` in CMakeLists.txt files.