Simplify your code

Kyle J. Knoepfel
24 June 2019
LArSoft Workshop 2019
“Keep it simple” … ?
“Keep it simple” … ?

- Nobody intentionally creates software to be complex, so why does it become so?
“Keep it simple” … ?

• Nobody intentionally creates software to be complex, so why does it become so?
  – The problems to be solved are complex…(not usually the cause)
  – Lack of knowledge or experience in designing software.
  – Lack of discipline.
  – Lack of time to clean things up.
  – etc.
“Keep it simple” … ?

• Nobody intentionally creates software to be complex, so why does it become so?
  – The problems to be solved are complex…(not usually the cause)
  – Lack of knowledge or experience in designing software.
  – Lack of discipline.
  – Lack of time to clean things up.
  – etc.

• 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 hard to understand
“Keep it simple” … ?

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

• LArSoft contributors often add code, but rarely remove it.

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

• For this talk I will focus primarily on simplifications, not conventions.
“I would have written a shorter letter, but I did not have the time.”

- Blaise Pascal
“I would have written a shorter letter, but I did not have the time.”
- Blaise Pascal

“It takes a lot of hard work to make something simple.”
- Steve Jobs
“I would have written a shorter letter, but I did not have the time.”
- Blaise Pascal

“It takes a lot of hard work to make something simple.”
- Steve Jobs
• Please, read this!
• Fermilab library has a few copies.
• Almost 15 years old; still relevant today.
• Please, read this!
• Fermilab library has a few copies.
• Almost 15 years old; still relevant today.

• Rule 6: Correctness, simplicity, and clarity come first.

“Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.”
- Alan Perlis

“The importance of a simple design cannot be overemphasized.”
- Jon Bentley
Estimating LArSoft’s complexity level

• Various metrics of estimating how complicated a body of code is.
• A simplistic one is counting lines of code.
Estimating LArSoft’s complexity level

- Various metrics of estimating how complicated a body of code is.
- A simplistic one is counting lines of code.

<table>
<thead>
<tr>
<th>Date</th>
<th>Tag</th>
<th>Lines of code</th>
<th>Lines of comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-08-11</td>
<td>v06_03_00</td>
<td>251138</td>
<td>95712</td>
</tr>
<tr>
<td>2017-07-25</td>
<td>v06_45_00</td>
<td>289929</td>
<td>111898</td>
</tr>
<tr>
<td>2018-07-30</td>
<td>v07_00_00</td>
<td>347079</td>
<td>137420</td>
</tr>
<tr>
<td>2019-06-08</td>
<td>v08_22_00</td>
<td>354545</td>
<td>141201</td>
</tr>
</tbody>
</table>

\(^1\) As computed by the cloc utility.
Estimating LArSoft’s complexity level

- Various metrics of estimating how complicated a body of code is.
- A simplistic one is counting lines of code.

<table>
<thead>
<tr>
<th>Date</th>
<th>Tag</th>
<th>Lines of code(^1)</th>
<th>Lines of comment(^1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-08-11</td>
<td>v06_03_00</td>
<td>251138</td>
<td>95712</td>
</tr>
<tr>
<td>2017-07-25</td>
<td>v06_45_00</td>
<td>289929</td>
<td>111898</td>
</tr>
<tr>
<td>2018-07-30</td>
<td>v07_00_00</td>
<td>347079</td>
<td>137420</td>
</tr>
<tr>
<td>2019-06-08</td>
<td>v08_22_00</td>
<td>354545</td>
<td>141201</td>
</tr>
</tbody>
</table>

\(^1\) As computed by the cloc utility.

- How do we reduce the maintenance burden?
Remove unnecessary files

• Remove files *that you know* are not needs. This may take approval from the collaboration.
  – LArSoft took these types of steps last week.

• 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 a .cc file (only .cc needed)
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.

<table>
<thead>
<tr>
<th>Built code</th>
<th>Build time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire file</td>
<td>11.3 s</td>
</tr>
<tr>
<td>Only headers</td>
<td>8.0 s</td>
</tr>
<tr>
<td>Only <em>art</em> headers</td>
<td>5.0 s</td>
</tr>
<tr>
<td>Empty file</td>
<td>0.4 s</td>
</tr>
</tbody>
</table>

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.
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. So be it.

- Only include headers in the file that are required for that file.
  - No courtesy headers!

<table>
<thead>
<tr>
<th>Discouraged</th>
<th>Encouraged</th>
</tr>
</thead>
</table>
| // MyService.h  
// The following headers are used  
#include <vector>  

// The following headers are not used  
#include "mf/.../MessageLogger.h"  
#include "art/.../ServiceHandle.h"  | // MyService.h  
// The following headers are used  
#include <vector>  |
Remove unnecessary link-time dependencies

- The SimWire test from earlier:

<table>
<thead>
<tr>
<th>Built code</th>
<th>Build time¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire file</td>
<td>11.3 s</td>
</tr>
<tr>
<td>Only headers</td>
<td>8.0 s</td>
</tr>
<tr>
<td>Only art headers</td>
<td>5.0 s</td>
</tr>
<tr>
<td>Empty file</td>
<td>0.4 s</td>
</tr>
</tbody>
</table>

- All steps included linking time. If we reduce the number of linked libraries…
Remove unnecessary link-time dependencies

- The SimWire test from earlier:

<table>
<thead>
<tr>
<th>Built code</th>
<th>Build time¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire file</td>
<td>11.3 s</td>
</tr>
<tr>
<td>Only headers</td>
<td>8.0 s</td>
</tr>
<tr>
<td>Only art headers</td>
<td>5.0 s</td>
</tr>
<tr>
<td>Empty file</td>
<td>0.4 s</td>
</tr>
<tr>
<td>Empty file + only art libraries</td>
<td>0.3 s</td>
</tr>
</tbody>
</table>

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

- And then:

```cpp
MyProducer::~MyProducer() {}
void MyProducer::beginJob() {}
void MyProducer::endJob() {}
```
Remove unnecessary functions

- If there is no work to be done in the following functions, remove them:
  - beginJob
  - beginRun
  - beginSubRun
  - endSubRun
  - endRun
  - endJob
  - Destructor
Remove unnecessary functions

- If there is no work to be done in the following functions, remove them:
  - beginJob
  - beginRun
  - beginSubRun
  - endSubRun
  - endRun
  - endJob
  - Destructor

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

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

- If there is no work to be done in the following functions, remove them:
  - beginJob
  - beginRun
  - beginSubRun
  - endSubRun
  - endRun
  - endJob
  - Destructor

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

private:
  void produce(art::Event&) override;
- void beginJob() override;
- void endJob() override;
};
```
To reconfigure or not to reconfigure…

- Consider this code:

```cpp
class MyProducer {
    LargeObject obj_;  // LargeObject
    unsigned counter_;  // unsigned integer
    unsigned importantConstant_;  // unsigned integer

public:
    MyProducer(ParameterSet const& pset) {
        reconfigure(pset);
    }

    void reconfigure(ParameterSet const& p) {
        obj_ = LargeObject{p.get<std::string>("some_label")};
        counter_ = 0;
        importantConstant_ = 42;
    }
};
```
To reconfigure or not to reconfigure…

• Consider this code:

```cpp
class MyProducer {
  LargeObject obj_; // Only const access needed
  unsigned counter_;
  unsigned importantConstant_; // Only const access needed

public:

  MyProducer(ParameterSet const& pset)
  {
    reconfigure(pset);
  }

  void reconfigure(ParameterSet const& p)
  {
    obj_ = LargeObject{p.get<std::string>("some_label")};
    counter_ = 0;
    importantConstant_ = 42;
  }
};
```
To reconfigure or not to reconfigure…

- Consider this code:

```cpp
class MyProducer {
  LargeObject obj_; // Only const access needed
  unsigned counter_;  
  unsigned importantConstant_; // Only const access needed

public:
  MyProducer(ParameterSet const& pset)
  {
    reconfigure(pset);
  }

  void reconfigure(ParameterSet const& p)
  {
    obj_ = LargeObject{p.get<std::string>("some_label")};
    counter_ = 0;
    importantConstant_ = 42;
  }
};
```

LargeObject() called before reconfigure is called
To reconfigure or not to reconfigure…

• Consider this code:

```cpp
class MyProducer {
    LargeObject obj_; // Only const access needed
    unsigned counter_; // Only const access needed
    unsigned importantConstant_; // Only const access needed

public:
    MyProducer(ParameterSet const& pset)
    {
        reconfigure(pset);
    }

    void reconfigure(ParameterSet const& p)
    {
        obj_ = LargeObject{p.get<std::string>("some_label")};
        counter_ = 0;
        importantConstant_ = 42;
    }
};
```

LargeObject() called before reconfigure is called

LargeObject(string const&) called
To reconfigure or not to reconfigure…

Consider this code:

```cpp
class MyProducer {
    LargeObject obj_; // Only const access needed
    unsigned counter_; // Only const access needed
    unsigned importantConstant_; // Only const access needed

public:
    MyProducer(ParameterSet const& pset)
    {
        reconfigure(pset);
    }

    void reconfigure(ParameterSet const& p)
    {
        obj_ = LargeObject{p.get<std::string>("some_label")};
        counter_ = 0;
        importantConstant_ = 42;
    }
};
```

**To boot:** module reconfiguration is not supported by art
To reconfigure or not to reconfigure...

Consider this code:

```cpp
class MyProducer {
    LargeObject obj_;  
    unsigned counter_; 
    unsigned importantConstant_;  
  public:
      MyProducer(ParameterSet const& pset) 
      {
          reconfigure(pset);  
      }
      void reconfigure(ParameterSet const& p) 
      {
          obj_ = LargeObject{p.get<std::string>("some_label")};
          counter_ = 0;
          importantConstant_ = 42;
      }
};
```

*LargeObject( ) called before reconfigure is called*

*LargeObject(string const& ) called*

Use the class’s initialization list!
To reconfigure or not to reconfigure…

Using the initialization list

class MyProducer {
    LargeObject obj_;
    unsigned counter_;
    unsigned importantConstant_;

public:

    MyProducer(ParameterSet const & pset)
        : obj_{p.get<std::string>("some_label")},
          counter{0},
          importantConstant_{42}
    {}    
};
To reconfigure or not to reconfigure…

Using the initialization list
• obj_ is constructed once

```cpp
class MyProducer {
  LargeObject obj_;  
  unsigned counter_;  
  unsigned importantConstant_; 

public:

  MyProducer(ParameterSet const& pset)
    : obj_{p.get<std::string>("some_label")},
      counter{0},
      importantConstant_{42}
  {}  
};
```
To reconfigure or not to reconfigure…

Using the initialization list
- `obj_` is constructed once
- `obj_` and `importantConstant_` can now be const

```cpp
class MyProducer {
    LargeObject const obj_;  
    unsigned counter_;        
    unsigned const importantConstant_;  

public:
    MyProducer(ParameterSet const& pset)
        : obj_(p.get< std::string >("some_label"))  
          , counter_{0}
          , importantConstant_{42}
    {};}
```
To reconfigure or not to reconfigure…

Using the initialization list
- obj_ is constructed once
- obj_ and importantConstant_ can now be const
- Use default values to reduce the number of required arguments

```cpp
class MyProducer {
    LargeObject const obj_;
    unsigned counter_{0};
    unsigned const importantConstant_{42};

public:
    MyProducer(ParameterSet const& pset)
        : obj_{p.get<std::string>("some_label")}
    {
    }
};
```
To reconfigure or not to reconfigure…

Using the initialization list

- obj_ is constructed once
- obj_ and importantConstant_ can now be const
- Use default values to reduce the number of required arguments

```cpp
class MyProducer {
  LargeObject const obj_;  // Constructor initializes obj_.
  unsigned counter_{0};   // Public member variable.

  unsigned const importantConstant_{42};
  public:
    MyProducer(ParameterSet const& pset) : obj_{p.get<std::string>("some_label")} {}  // Using initialization list.
};
```

Get rid of module reconfigure functions.
Remove inappropriate preprocessor use

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

• Header guards are for headers!
  – Do not place header guards in implementation (.cc) files.

• Do not define constants with macros
  – BAD:  #define NUM_BEETHOVEN_SYMPHONIES 9
  – GOOD: constexpr unsigned int num_beethoven_symphonies{9};

• ROOT no longer supports the ___GCCXML___ preprocessor guard. If you absolutely need to hide code from the dictionary generator, use ___ROOTCLING___.

More simplifications

- Defining *art* modules

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

- Iterating over `std::map` entries

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

• Creating `std::unique_ptr`s

```cpp
- 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

```cpp
- namespace a {
-   namespace b {
-     ...
-   }
- }
+ namespace a::b {
+   ...
+ }
```
LArSoft’s coupling to *art*

- Much of LArSoft has been built on top of *art* and *canvas*
- This makes sense for the components that are meant to interact with a framework
- LArSoft provides facilities that are not intrinsically connected to any framework
  - I encourage you to reduce your reliance on *art*- or *canvas*-provided interface.
  - It is a maintenance burden, and who knows where frameworks will be *n* years from now
LArSoft’s coupling to art

• Much of LArSoft has been built on top of art and canvas

• This makes sense for the components that are meant to interact with a framework

• LArSoft provides facilities that are not intrinsically connected to any framework
  – I encourage you to reduce your reliance on art- or canvas-provided interface.
  – It is a maintenance burden, and who knows where frameworks will be n years from now

• Practical suggestion: no ServiceHandles outside of art-supported plugins
  – Providers should never create ServiceHandles
  – Algorithms in larreco/RecoAlg should never create ServiceHandles
  – etc.
Takeaways

• Making things simpler takes a lot of effort.

• Ways to get there:
  – Remove unnecessary files
  – Remove unnecessary header dependencies
  – Remove unnecessary link-time dependencies
  – Remove unnecessary functions/classes
  – Use modern C++ facilities to simplify your code
  – Reduce coupling to art

• Come talk to the SciSoft team. We’re here to help you.