Fermilab | U.S. DEPARTMENT OF ENERGY | Office of Science

art 2.06.01 and tools

Kyle J. Knoepfel
14 February 2017

art 2.06.01

- Released last week.
- New features:
  - Extended `MixFilter` abilities for `Events` and their associated (`Sub`)`Run` products
  - Relaxed `art::Assns<A,B>` lookup policy and relaxed `art::Ptr<T>` resolution rules for smart query objects
  - **Introduction of the *art* tool**
- For more information, see the release notes at:
  - *art* 2.06.00
  - *art* 2.06.01

What is an *art* tool?

Description by analogy:

- A **module** makes it possible to modify a framework program's behavior without rebuilding the **framework**.
- A **tool** makes it possible to modify a module's behavior without rebuilding the **module**.

What is an *art* tool?

Description by analogy:

- A **module** makes it possible to modify a framework program's behavior without rebuilding the **framework**.
- A **tool** makes it possible to modify a module's behavior without rebuilding the **module**.

Features of tools:

- Plugin libraries, dynamically loaded at run-time
- Which libraries to load are specified by the user's configuration
- The tools themselves are configurable
- A tool instance can be created anywhere in your code
- A tool instance is local *only* to the scope in which it was created (i.e. no global tool registry)

When to use an *art* tool

There are two criteria that should be satisfied before you consider using an *art* tool:

(a) The subtask to be done does not make sense outside of the context of the module, **and**

(b) The user needs to be able to extend what your module does without modifying the module code.

Why not use a service?

Two motivations for a service:

1. They provide hooks to the *art* state transitions (primary motivation).
2. They can provide global access to a given service instance via `ServiceHandle`. However, this is encouraged *only* for a service that has a `const`-only interface.

The need for the *art* tool is not addressed by either of these motivations.

Kinds of *art* tools

Function tool Loadable library that provides access to a free function (*e.g.* `RT myFunction(ARGS...)`). When a tool of this kind is created, its C++ type is `std::function<RT(ARGS...)>`.

Class tool Loadable library that is represented by a user-defined class (*e.g.* `MyInterface`). When a tool of this kind is created, its C++ type is `std::unique_ptr<MyInterface>`.

Recommendations:

- Favor a function tool over a class tool. Only choose a class tool if you need to retain state during interface calls.
- *Creating* a tool is encouraged only in the constructor of the module that uses it. It can be used elsewhere in the module, however.

Defining a function tool

```
1 // makeTracks.hh
2 // ... #includes
3 Tracks makeTracks(Hits const&);
```

Defining a function tool

```
// makeTracks.hh
// ... #includes
Tracks makeTracks(Hits const&);
```

Now create a source file from which a `*_tool.so` library is made.

```
// makeTracks_tool.cc
#include "art/Utilities/ToolMacros.h"
#include "makeTracks.hh"

Tracks makeTracks(Hits const& hits) {
  // Make 'tracks' from 'hits'...
  return tracks;
}
DEFINE_ART_FUNCTION_TOOL(makeTracks, "Trks")
```

Using a function tool

To make use of the tool, include the appropriate headers, including:

- `art/Utilities/make_tool.h`
- `makeTracks.hh`

```
std::function<decltype(makeTracks)> make_;
make_=make_tool<decltype(makeTracks)>(nps,"Trks");
```

Notice that the second argument to `art::make_tool` (`"Trks"`) agrees with the second argument provided to the `DEFINE_ART_FUNCTION_TOOL` macro on the previous slide.

Using a function tool

To make use of the tool, include the appropriate headers, including:

- `art/Utilities/make_tool.h`
- `makeTracks.hh`

```
std::function<decltype(makeTracks)> make_;
make_=make_tool<decltype(makeTracks)>(nps,"Trks");
```

Call function wherever it's needed:

```
void produce(art::Event& e) override
{
  auto const& hits = e.getValidHandle<Hits>(...);
  auto tracks = make_(*hits);
  e.put(make_unique<Tracks>(move(tracks)));
}
```

Specifying a function tool in your FHiCL file

Include a nested table in your module's configuration, which contains a parameter named `tool_type`, whose value is the *basename* of the `.so` file.

```
trackProducer: {
  module_type: TrackProducer
  trackAlgo: {
    tool_type: makeTracks
  }
}
```

From the previous page:

```
art::make_tool<decltype(makeTracks)>(nps, "Trks");
```

where `nps` is equal to:

```
module_ps.get<ParameterSet>("trackAlgo")
```

Defining a class tool

```
class Counter {
public:
  virtual ~Counter() noexcept = default;
  virtual void update(unsigned) = 0;
  virtual unsigned count() const = 0;
};
```

Defining a class tool

```
  // SimpleCounter_tool.cc
#include "art/Utilities/ToolMacros.h"
#include "Counter.hh"
class SimpleCounter : public Counter {
public:
  SimpleCounter(ParameterSet const& ps) :
    count_{ps.get<string>("offset")}
  {}
private:
  void update(unsigned n) override {count_ += n;}
  unsigned count() const override {return count_;}
  unsigned count_;
};
DEFINE_ART_CLASS_TOOL(SimpleCounter)
```

Using a class tool

```
std::unique_ptr<Counter> counter_;
counter_ = art::make_tool<Counter>(nps);
```

Header dependency required just for `Counter`, not for a derived type.

Using a class tool

```
std::unique_ptr<Counter> counter_;
counter_ = art::make_tool<Counter>(nps);
```

```
void analyze(art::Event const& e) override
{
  auto const& ts = e.getValidHandle<Tracks>(...);
  counter_->update(ts.size());
}
```

```
void endJob() override
{
  LogInfo(...) << "Number of tracks seen: "
               << counter_->count() << '\n';
}
```

Specifying a class tool in your FHiCL file

Include a nested table in your module's configuration:

```
trackCounter: {
  module_type: TrackCounter
  counterAlgo: {
    tool_type: SimpleCounter # Derived type
    offset: 0 # Additional parameters for tool
  }
}
```

From the previous page:

```
art::make_tool<decltype(makeTracks)>(nps);
```

where `nps` is equal to:

```
module_ps.get<ParameterSet>("counterAlgo")
```

Concluding remarks

- *art* tools are now supported as of version 2.06.00 and 2.06.01.
- They provide a means of adjusting the behavior of a module without having to rebuild it.
- It is possible to implement configuration validation and description with tools. Please get ahold of me, if you're interested.
- Documentation is still being written; however, please see:
  - General design considerations
  - Guide to writing and using tools