GSoC-2020 - Reduce boost dependence in CMSSW

Lucas Camolezi

Mentors
Vassil Vassilev
David Lange

Project idea

This project has the objective to reduce CMSSW technical debt by finding and replacing boost dependencies that have a equivalent solution in standard C++.

> “Lets try to take advantage of “new” language developments that were not available when CMSSW started”

First stage

Replaced boost dependencies that have a very similar counterpart in stl c++.

- boost::array → std::array
- boost::unordered_map → std::unordered_map
- boost::unordered_set → std::unordered_set
- boost::bind →std::bind or C++ lambdas
- boost::function → std::function
- boost::random → stl <random>
- boost::hash → std::hash
- boost::filesystem → std::filesystem
- boost::mutex → std::mutex
- boost::variant →std::variant

Exemple:
boost::bind

```
#include <sstream>
#include <sigc++/sigc++.h>
- #include <boost/bind.hpp>
+ #include <functional>
#include "TClass.h"
#include "TGFrame.h"
#include "TGTab.h"
@@ -70,7 +70,7 @@ CmsShowEDI::CmsShowEDI(const TGWindow* p, UInt_t w, UInt_t h, FWSelectionManager
   m_selectionManager = selMgr;
   SetCleanup(kDeepCleanup);

-   m_selectionManager->itemSelectionChanged_.connect(boost::bind(&CmsShowEDI::fillEDIFrame, this));
+   m_selectionManager->itemSelectionChanged_.connect(std::bind(&CmsShowEDI::fillEDIFrame, this));

   TGVerticalFrame* vf = new TGVerticalFrame(this);
   AddFrame(vf, new TGLayoutHints(kLHintsExpandX | kLHintsExpandY, 0, 0, 0, 0));
@@ -292,10 +292,10 @@ void CmsShowEDI::fillEDIFrame() {
       updateLayerControls();
       m_layerEntry->SetState(kTRUE);
       m_displayChangedConn =
-          m_item->defaultDisplayPropertiesChanged_.connect(boost::bind(&CmsShowEDI::updateDisplay, this));
-      m_modelChangedConn = m_item->changed_.connect(boost::bind(&CmsShowEDI::updateFilter, this));
+          m_item->defaultDisplayPropertiesChanged_.connect(std::bind(&CmsShowEDI::updateDisplay, this));
+      m_modelChangedConn = m_item->changed_.connect(std::bind(&CmsShowEDI::updateFilter, this));
       //    m_selectionChangedConn = m_selectionManager->selectionChanged_.connect(boost::bind(&CmsShowEDI::updateSelection, this));
-      m_destroyedConn = m_item->goingToBeDestroyed_.connect(boost::bind(&CmsShowEDI::disconnectAll, this));
+      m_destroyedConn = m_item->goingToBeDestroyed_.connect(std::bind(&CmsShowEDI::disconnectAll, this));

       clearPBFrame();
       m_item->getConfig()->populateFrame(m_settersFrame);
```

Second stage

Boost dependencies that don’t have a counterpart in stl. But, they still can be replaced cleanly using a minor workaround.

- boost::noncopyable
  - Solution: Delete copy constructor and operation
- boost::ptr_vector, boost::ptr_list, boost::ptr_map, boost::ptr_deque
  - Solution: boost::ptr_T<U> → std::T<std::unique_ptr<U>>
- boost::matrix
  - Solution: Wrapped std::vector
- boost::mpl
  - <type_traits>

Exemple: boost::noncopyable

```
-     struct SingleInvoker : boost::noncopyable {
+     struct SingleInvoker {
      private:  // Private Data Members
        method::TypeCode retType_;
        std::vector<MethodInvoker> invokers_;
@@ -72,6 +70,9 @@ namespace reco {
        bool isRefGet_;

      public:
+       SingleInvoker(const SingleInvoker&) = delete;
+       SingleInvoker& operator=(const SingleInvoker&) = delete;
+
        SingleInvoker(const edm::TypeWithDict&, const std::string& name, const std::
        ~SingleInvoker();
```

Exemple: boost::mpl

```
      };

-     typedef typename ::boost::mpl::if_<prod1, prod1, typename ::boost::mpl::if_<prod2, prod2, prod0>::type>::type prod;
+     typedef
+         typename std::conditional<prod1::value, prod1, typename std::conditional<prod2::value, prod2, prod0>::type>::type
+             prod;
      typedef typename AuxProductRatio2<prod>::type type;
      inline static type combine(const H& h, const PROD_S(F, G) & fg) {


  TEMPL(N2T1) struct SimpifyS2C2Sum<n, m, A, false> {
-     static const int p = ::boost::mpl::if_c<(n < m), ::boost::mpl::int_<n>, ::boost::mpl::int_<m> >::type::value;
+     static constexpr int p = n < m ? n : m;
```

Results

	Start of project (May 4)	Final (Sep 1)
Usage - git grep boost:: | wc -l	2875	1961
Includes - git grep boost/ | wc -l	1201	803

Things without clear replacements

Examples of remaining dependencies without a stl counterpart. These still are in cmssw.

- boost::algorithm (string) → we could implement our own version for simple algorithms.
- boost::format →std::format c++20.
- boost::posix →c++20 <chrono> probably could replace part of this.
- boost::regex -> std::regex low performance
- boost::serialization
- boost::iterator
- boost::lexical_cast

Motivation

```
#include <vector>
```

Textual Include

✗ Expensive
✗ Fragile

1. **Expensive**
   Reparse the same header
2. **Fragile**
   Name collisions

Rcpp library

```
#define PI 3.14
...
```

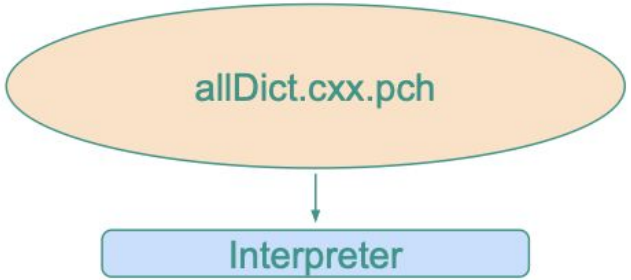
User Code

```
#include <header.h>
...
double PI = 3.14;
// => double 3.14 = 3.14;
```

Precompiled Headers (PCH)

✗ Inseparable

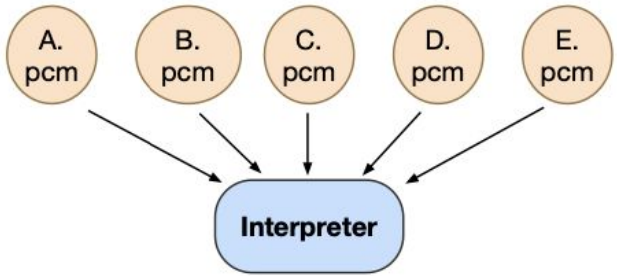
1. Storing precompiled header information (same as modules)
2. Stored in one big file
   - Inseparable



Modules

✓

- Pre compiled PCM files contain header information
- PCMs are separated



Modules C++ 20

Classical header model

Compile-time scalability: Each time a header is included, the compiler must preprocess and parse the text in that header and every header it includes.

Fragility: #include directives are treated as textual inclusion by the preprocessor, and are therefore subject to any collision with a name in the library with macros.

Tool confusion: In a C-based language, it is hard to build tools that work well with software libraries, because the boundaries of the libraries are not clear. Which headers belong to a particular library, and in what order should those headers be included to guarantee that they compile correctly?

modules

Compile-time scalability: A module is only compiled once, and importing the module into a translation unit is a constant-time operation (independent of module system)

Fragility: Each module is parsed as a standalone entity, so it has a consistent preprocessor environment. This completely eliminates the need for __underscored names and similarly defensive tricks. Also eliminating include-order dependencies.

Tool confusion: Modules describe the API of software libraries, and tools can reason about and present a module as a representation of that API. Because modules can only be built standalone, tools can rely on the module definition to ensure that they get the complete API for the library.

boost modules (clang c++20)

C++ modules require each header file to compile on its own. That is, all needed headers need to be directly included rather than relying on accidental indirect includes.

We found that some boost headers needed to be updated to meet this criterion.

- We can change the boost headers that need a “fix” and submit to upstream boost.

Reducing boost dependencies:

- Reducing boost dependencies helps us create more lightweight boost clang modules for upcoming c++20.
- This also reduces the amount of headers that we need to work on to be able to use c++20 clang modules.

Results so far

- **85** boost modules built.

- **57** patches to headers in boost.