

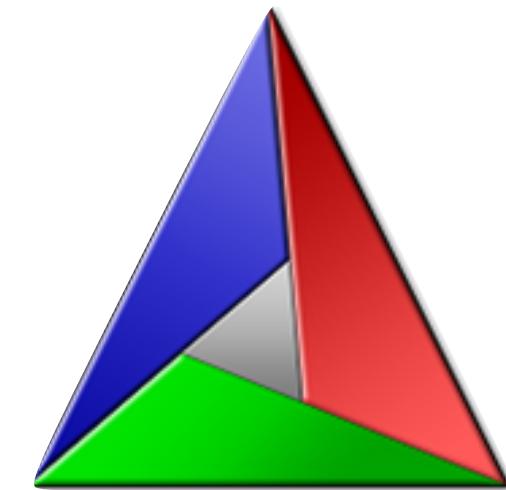


CMake

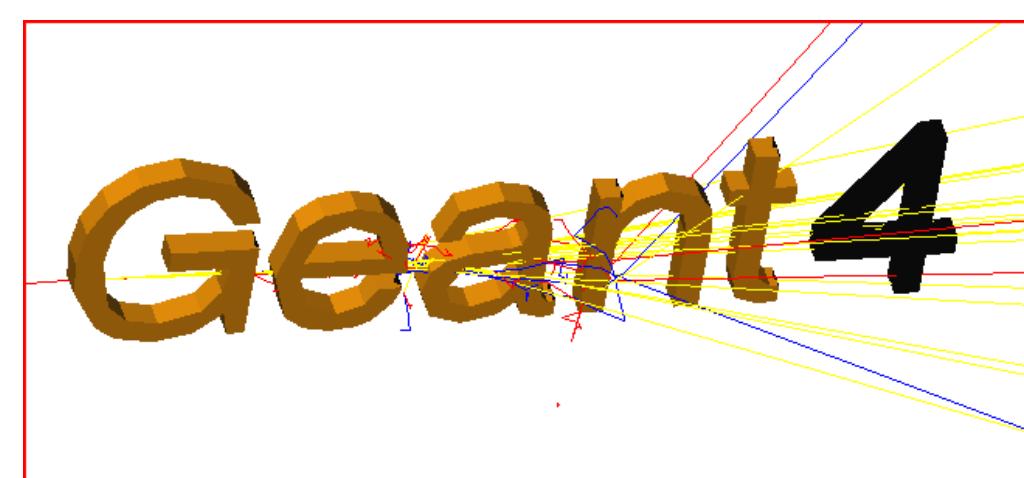
ATLAS's (Not So New) Software Build Tool

Attila Krasznahorkay

CMake



CMake
Cross-platform Make



- ATLAS decided years ago to replace CMT (and RootCore) with a more modern build system
 - Mainly to stop using systems that nobody else was using...
 - The final choice was CMake
- Adopted by many prominent projects as their build system by now
 - Including an ever growing number inside HEP as well
- A general build system, **not** specifically designed for ATLAS
 - Prompting the examination of some of our workflows



Code Compilation

- For the simplest setup, like a `helloWorld.cxx` source file, you don't need any help for the compilation

```
g++ -o helloWorld helloWorld.cxx
```

- Once you have multiple source files, you don't want to execute such commands one by one anymore
 - This is where GNU Make comes in, allowing you to describe the build procedure of your files in a very low level way

```
myExecutable: source1.o source2.o
    g++ -o $@ $^
.SUFFIXES: .cxx .o
.cxx.o:
    g++ -c -o $@ $<
```

- As your project grows, you start depending on more exotic “externals”, GNU Make doesn't scale anymore
 - You need a higher level build system at that point

Build System Generator



- CMake doesn't actually perform the build itself
 - It generates a configuration for some underlying build system to do it
 - On UNIX in most cases it is GNU Make
 - However Ninja is becoming more and more powerful/popular, and is also capable of building the full ATLAS offline software
 - But it can be an IDE as well. Like Xcode on Mac OS X, or Visual Studio on Windows
- Once the build system's configuration is generated, you just interact with that build system to perform the build

Config - Build - Install



- Like all autoconf / automake projects, CMake projects also follow these 3 steps during the build

- Configure
 - Either in-source

```
cd project-1.2.3
cmake -DCMAKE_INSTALL_PREFIX=/usr/local
```

- Or out-of-source

```
mkdir build; cd build/
cmake -DCMAKE_INSTALL_PREFIX=/usr/local ..../project-1.2.3
```

- Build

```
make -jX
```

- Install

```
[sudo] make install [DESTDIR=/some/location]
```

Config - Build - Install

- Like all autoconf / automake projects, CMake projects also follow these 3 steps during the build

- Configure
 - Either in-source

Optional Configuration

```
cd project-1.2.3
cmake -DCMAKE_INSTALL_PREFIX=/usr/local
```

- Or out-of-source

```
mkdir build; cd build/
cmake -DCMAKE_INSTALL_PREFIX=/usr/local ..../project-1.2.3
```

- Build

```
make -jX
```

- Install

```
[sudo] make install [DESTDIR=/some/location]
```

Config - Build - Install

- Like all autoconf / automake projects, CMake projects also follow these 3 steps during the build

- Configure
 - Either in-source

```
cd project-1.2.3
cmake -DCMAKE_INSTALL_PREFIX=/usr/local
```

Optional Configuration

- Or out-of-source

```
mkdir build; cd build/
cmake -DCMAKE_INSTALL_PREFIX=/usr/local ..../project-1.2.3
```

**Always use this with
ATLAS code**

- Build

```
make -jX
```

- Install

```
[sudo] make install [DESTDIR=/some/location]
```

Config - Build - Install

- Like all autoconf / automake projects, CMake projects also follow these 3 steps during the build

- Configure
 - Either in-source

```
cd project-1.2.3
cmake -DCMAKE_INSTALL_PREFIX=/usr/local
```

Optional Configuration

- Or out-of-source

```
mkdir build; cd build/
cmake -DCMAKE_INSTALL_PREFIX=/usr/local ..../project-1.2.3
```

**Always use this with
ATLAS code**

- Build

```
make -jX
```

- Install

```
[sudo] make install [DESTDIR=/some/location]
```

**In “local builds”
usually not necessary**

AtlasCMake / AtlasLCG



- As said, CMake itself is a very general language for describing the build of some software
- ATLAS does have very concrete conventions on how it builds its code though
 - All of the helper code for building ATLAS packages is held by the atlas/atlasexternals repository
 - Most of the CMake logic is in the AtlasCMake and AtlasLCG subdirectories of the repository...
 - You can find some useful information about the code just by browsing the Git repository
- For a reference documentation of AtlasCMake's features, see:
<https://twiki.cern.ch/twiki/bin/viewauth/AtlasComputing/SoftwareDevelopmentWorkBookCMakeInAtlas>

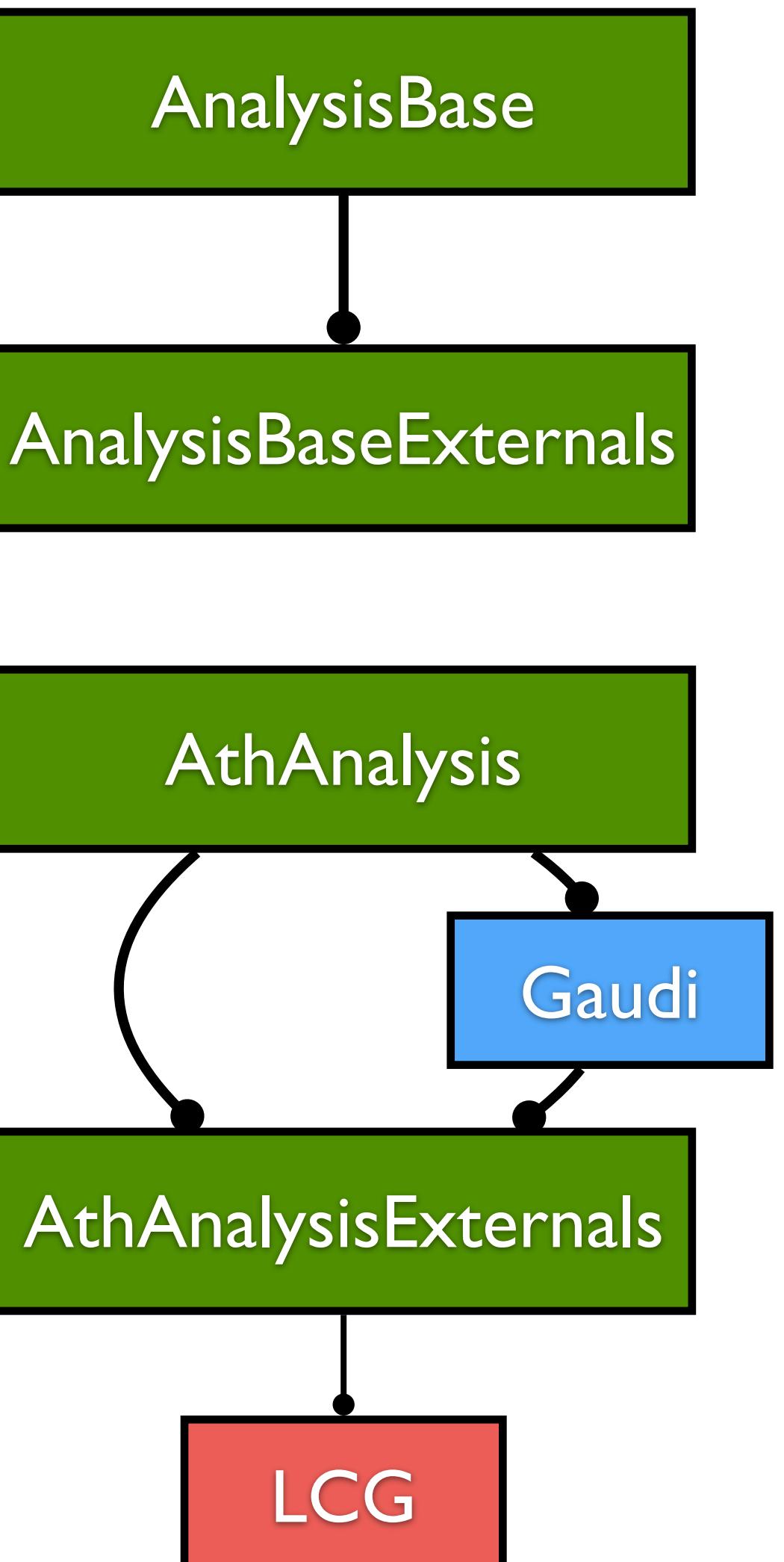


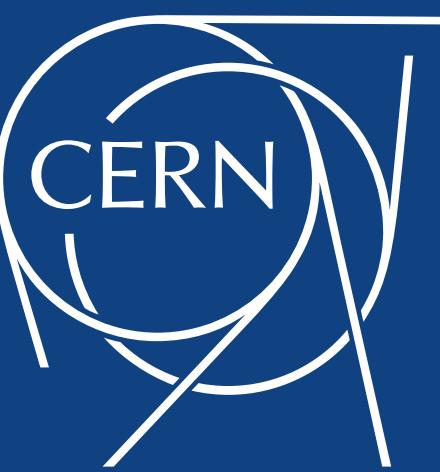
Packages

- Historically ATLAS code is organised into “packages”
 - Distinct build units that have their own configuration for how they should be built
- With CMake this is not the most natural organisation, but we kept it not to change too many things at once
- The layout of any ATLAS project is hence:
 - A `CMakeLists.txt` project configuration either at the root directory of the source tree, or in a subdirectory of the `Projects/` folder
 - Many sub-directories with a `CMakeLists.txt` file of their own
 - Note that package configuration files are not functional on their own! They only work correctly when a project configuration file includes them “in the right way”.

Projects

- In order to provide external software for our builds in a well controlled way, every ATLAS CMake project starts from a base project
 - These are defined in [atlasexternals/Projects](#)
- For regular code development you should not worry about these externals projects
 - But if you want to introduce the usage of some new external library for instance, then you need to know about them





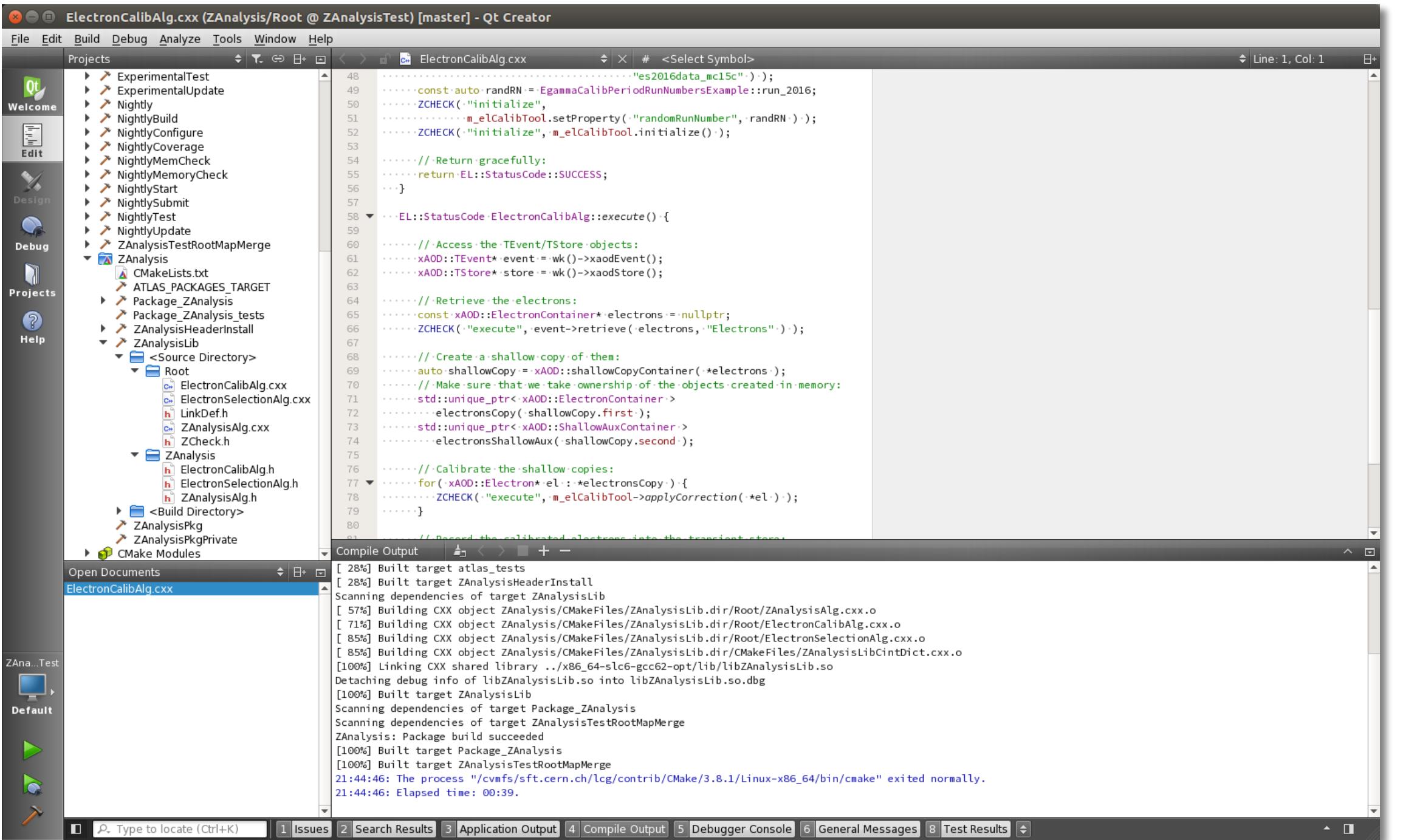
Building Projects

- Covered in good detail on <https://atlassoftwaredocs.web.cern.ch>
- The main points:
 - ATLAS repositories can all build multiple projects. So you don't point CMake at the repository's root directory, but at one of the directories in the Projects/ folder.
 - When working with the athena repository, use the Projects/WorkDir project to build a selected number of packages against a release/nightly.
 - You can build entire projects from scratch using the build scripts in their project directories. Instructions for the scripts can be found in the individual project directories.
 - For instance AnalysisBase can be built from the 21.2 branch of atlas/athena with:

```
git clone ...
./athena/Projects/AnalysisBase/build_externals.sh
./athena/Projects/AnalysisBase/build.sh
```

Summary

- CMake is the build system for all ATLAS projects
- Whatever you learn in this tutorial is directly applicable in offline/trigger software development as well
- For further instructions on good code layout with CMake and Git, see the upcoming talks



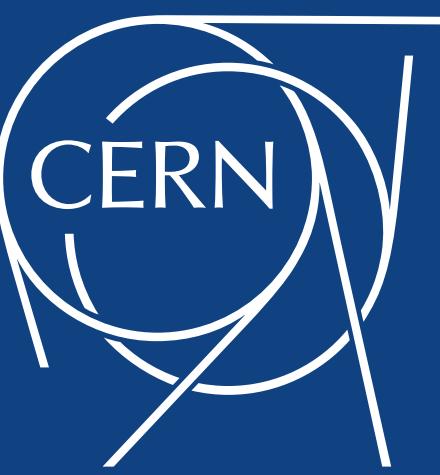
The screenshot shows the Qt Creator IDE interface. The main window displays the code for `ElectronCalibAlg.cxx`. The code includes comments and several calls to `ZCHECK` and `m_elCalibTool`. The left sidebar shows the project structure under `ZAnalysis`, including `CMakeLists.txt`, `ATLAS_PACKAGES_TARGET`, and various source files like `ElectronCalibAlg.h` and `LinkDef.h`. The bottom panel shows the `Compile Output` tab, which displays the build log for the target `atlas_tests`. The log shows the compilation of multiple CXX objects and the linking of shared libraries, concluding with the message "The process ... exited normally."

```

48     .....const auto randRN = EgammaCalibPeriodRunNumbersExample::run_2016;
49     .....ZCHECK("initialize",
50     .....    _m_elCalibTool.setProperty("randomRunNumber", randRN));
51     .....ZCHECK("initialize", _m_elCalibTool.initialize());
52
53     .....// Return gracefully:
54     .....return EL::StatusCode::SUCCESS;
55
56 }
57
58 EL::StatusCode ElectronCalibAlg::execute() {
59
60     .....// Access the TEvent/Tstore objects:
61     .....xAOOD::TEvent* event = wk()->xaoEvent();
62     .....xAOOD::Tstore* store = wk()->xaoStore();
63
64     .....// Retrieve the electrons:
65     .....const xAOOD::electronContainer* electrons = nullptr;
66     .....ZCHECK("execute", event->retrieve(electrons, "Electrons"));
67
68     .....// Create a shallow copy of them:
69     .....auto shallowCopy = xAOOD::shallowCopyContainer(*electrons);
70     .....// Make sure that we take ownership of the objects created in memory:
71     .....std::unique_ptr<xAOOD::ElectronContainer>
72         electronsCopy(shallowCopy.first);
73     .....std::unique_ptr<xAOOD::ShallowAuxContainer>
74         electronsShallowAux(shallowCopy.second);
75
76     .....// Calibrate the shallow copies:
77     .....for( xAOOD::Electron* el : *electronsCopy ) {
78         .....ZCHECK("execute", _m_elCalibTool->applyCorrection(*el));
79     }
80
81     .....// Detach the calibrated electrons into the transient store:
82 }
```

[284] Built target atlas_tests
[284] Built target ZanalysisHeaderInstall
Scanning dependencies of target ZanalysisLib
[576] Building CXX object ZAnalysis/CMakeFiles/ZanalysisLib.dir/Root/ZanalysisAlg.cxx.o
[716] Building CXX object ZAnalysis/CMakeFiles/ZanalysisLib.dir/Root/ElectronCalibAlg.cxx.o
[856] Building CXX object ZAnalysis/CMakeFiles/ZanalysisLib.dir/Root/ElectronSelectionAlg.cxx.o
[856] Building CXX object ZAnalysis/CMakeFiles/ZanalysisLib.dir/Root/ZanalysisLibCintDict.cxx.o
[1006] Linking CXX shared library/lib/libAnalysisLib.so
Detaching debug info of libAnalysisLib.so into libAnalysisLib.so.dbg
[1006] Built target ZanalysisLib
Scanning dependencies of target Package_ZAnalysis
Scanning dependencies of target ZanalysisTestRootMapMerge
ZAnalysis: Package build succeeded
[1006] Built target Package_ZAnalysis
[1006] Built target ZanalysisTestRootMapMerge
21:44:46: The process "/cvmfss/sft.cern.ch/lcg/contrib/CMake/3.8.1/Linux-x86_64/bin/cmake" exited normally.
21:44:46: Elapsed time: 00:39.

Extra Material



PUBLIC / PRIVATE

- Often a source of confusion...
- In the end it's only relevant for shared libraries
 - Interface libraries only have public dependencies
 - Component libraries only have private ones
 - The “package dependencies” will go away at one point...
- When a shared library need an include path or a library as a public dependency, they will be used while compiling/linking the sources of the shared library, and also while compiling/linking any component linking against this shared library.
 - If an include is made to a package/shared library in a public header of this library, that's a public dependency.
- Any include paths or libraries that a shared library depends on privately will only be used while compiling/linking the shared library. They are ignored when building components linking against it.
 - If an include is only made to another shared library in a private source file, that's a private dependency.
- CMake even defines a 3rd dependency type, but we don't use that explicitly in the ATLAS code...



Library Types

- **Shared library**
 - Created by `atlas_add_library(...)`
 - Can be linked against, symbols defined in it can be used by other components
 - Has to define public headers
- **Component/module library**
 - Created by `atlas_add_component(...)`
 - Can not be linked against
 - Provides no public headers
 - Houses the Gaudi components of the software
- **Interface library**
 - Created by `atlas_add_library(...)` when using the INTERFACE argument with it
 - Provides an interface to headers. Very useful for header-only packages, and when a package provides both a component library and some public (interface) headers.
- **Some other types for T/P, POOL and BS converter libraries that you'll learn about if/when needed**
 - See: https://twiki.cern.ch/twiki/bin/viewauth/AtlasComputing/SoftwareDevelopmentWorkBookCMakeInAtlas#Generic_build_target_declaration

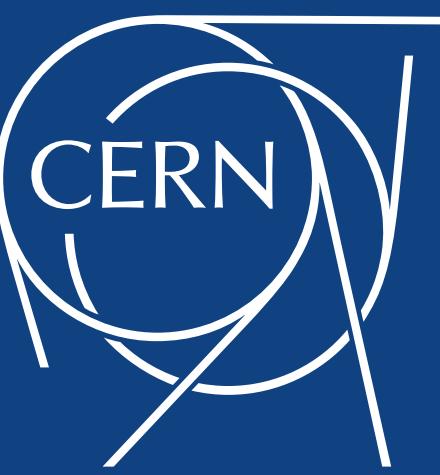


Using an IDE

- Two types of IDEs exist:
 - Ones interacting directly with CMake;
 - You can usually just open the project `CMakeLists.txt` file of your source area, and the IDE sets up itself correctly
 - Includes: [KDevelop](#), [QtCreator](#), [CLion](#), etc.
 - Ones not doing that.
 - You need to instruct CMake to generate a native project configuration for the IDE
 - Includes: [Eclipse](#), [Xcode](#), [Visual Studio](#), [Visual Studio Code](#), etc.
- On SLC6/lxplus I had the most success with [QtCreator](#)
 - If you want to give it a try, after setting up your build environment (from which you could call CMake to set up your build directory), execute:

```
/afs/cern.ch/work/k/krasznaa/public/QtCreator/4.6.2/x86_64-slc6-gcc62-opt/bin/qtcreator
```

- ...and point the application to the “main” `CMakeLists.txt` file of your code
 - I.e. File → Open...



Using an IDE

- Two types of IDEs exist:
 - Ones interacting directly with CMake;
 - You can usually just open the project `CMakeLists.txt` file of your source area, and the IDE sets up itself correctly
 - Includes: [KDevelop](#), [QtCreator](#), [CLion](#), etc.
 - Ones not doing that.
 - You need to instruct CMake to generate a native project configuration for the IDE
 - Includes: [Eclipse](#), [Xcode](#), [Visual Studio](#), [Visual Studio Code](#), etc.
- On SLC6/lxplus I had the most success with [QtCreator](#)
 - If you want to give it a try, after setting up your build environment (e.g. `lxplus` in your build directory), execute:

```
/afs/cern.ch/work/.../lxplus$ /afs/cern.ch/work/.../lxplus/QtCreator/4.6.2/x86_64-slc6-gcc62-opt/bin/qtcreator
```

- ...and point the application to the “main” `CMakeLists.txt` file of your code
 - I.e. File → Open...



Example Package Configuration

```
# Set the name of the package:  
atlas_subdir( MyPackage )  
  
# Set which other packages it depends on:  
atlas_depends_on_subdirs(  
    PRIVATE GaudiKernel Event/xAOD/xAODEgamma  
    Control/AthenaBaseComps )  
  
# Make use of TBB directly in the code:  
find_package( TBB )  
  
# Build a component library from the package's code:  
atlas_add_component( MyPackage  
    src/*.h src/*.cxx src/components/*.cxx  
    INCLUDE_DIRS ${TBB_INCLUDE_DIRS}  
    LINK_LIBRARIES ${TBB_LIBRARIES} GaudiKernel xAODEgamma AthenaBaseComps )  
  
# Install extra files from the package:  
atlas_install_python_modules( python/*.py )  
atlas_install_joboptions( share/*.py )
```



<http://home.cern>