

[TWiki](#) > [AtlasComputing Web](#) > [AtlasComputing](#) > [ComputingTutorials](#) > [SoftwareTutorial](#) > [SoftwareTutorialAdvancedCMake](#)  
(2017-11-22, JonathanThomasBurr)

## (An Advanced) Introduction to CMake

### [CMake Basics](#)

[Acquiring CMake](#)

[Configure - Build - Install](#)

[Separate Source and Build Directories](#)

### [CMake 101](#)

[Hello World!](#)

[Library and Executable](#)

[Building the Library](#)

[Building the Executable](#)

[Component Properties](#)

[Installing Targets and Files](#)

### [Semi-advanced CMake](#)

[Finding Externals](#)

[A Note on ROOT libraries](#)

[Exporting Your Project](#)

[Making Use of This Project in a Second One](#)

[Generator Expressions](#)

[Configuration Control Flow](#)

[Functions and Macros](#)

[Subdirectories](#)

[IDEs](#)

### [CMake in ATLAS](#)

[Setting Up the Environment](#)

[Checkout and Build](#)

[Package Configuration Files](#)

[Package Name and Dependencies](#)

[Building Libraries and Executables](#)

[Installing Files](#)

[Building and Running \(Unit\) Tests](#)

[Runtime Environment](#)

### [Resources](#)

This page provides an exhaustive description of CMake. How one can use it to build code completely independent of ATLAS, how we make use of it for building ATLAS software, and how one can use the features provided by the ATLAS CMake code to build code against ATLAS analysis (or offline) releases.

## CMake Basics

CMake is a high-level language for describing the build instructions for a piece of software. It is developed by [Kitware](#), and can be accessed on <https://cmake.org>. It is used as the build system of software products of many types and sizes. Including the software provided by Netflix, KDE, or ROOT.

CMake itself is not responsible for running the tools that build software. It's not a replacement of GNU Make and similar tools. Instead it's a layer on top of these tools. CMake is responsible for generating the proper configuration for lower level tools for actually building a software project. On UNIX systems this by default means generating makefiles that take care of the build itself. But CMake supports a large set of backends. Including a large set of different IDEs.

## Acquiring CMake

Practically all linux distributions provide a native package for CMake. On such systems your best bet is usually just to use the native `cmake` executable. Keep in mind though that the ATLAS CMake code requires at least CMake version 3.2 these days. If your platform's native CMake version is older than this, you should install a newer version by hand.

You can always find the sources and various binaries for the latest version(s) of CMake on their main download page:

<https://cmake.org/download/>. (3.6.2 at the time of writing.) The CMake binaries are very self-contained, so in most cases just downloading an appropriate binary is perfectly acceptable. But on UNIX-like systems building CMake is also very simple. You can do it like:

```
wget https://cmake.org/files/v3.6/cmake-3.6.2.tar.gz
tar -xvzf cmake-3.6.2.tar.gz
cd cmake-3.6.2/
./bootstrap --prefix=/usr/local/cmake/3.6.2
make -jX
[sudo] make install
```

Note that the `cmake` executable that in this case ends up under `/usr/local/cmake/3.6.2/bin/cmake` doesn't need any environment setup to run. You can just execute it with its full path. Making it very easy to test multiple versions of CMake on a single machine.

## Configure - Build - Install

CMake follows the same general idea as practically all UNIX projects. Very often you build a software package like:

```
tar -xvzf SomeSoftware-1.2.3.tar.gz
cd SomeSoftware-1.2.3/
./configure --prefix=/usr/local/SomeSoftware/1.2.3
make
[sudo] make install
```

Even the build of CMake itself follows the very same procedure.

This is exactly the same for CMake based projects. In general you take 3 steps to install them:

- Configure the build
- Build
- Install the products

As a reminder, CMake itself is really only used in the first step of all of this. During the project's configuration it generates a configuration for the underlying build system (GNU Make in most cases) that takes care of the second and third steps. All in all, in general you build a CMake based project like:

```
tar -xvzf SomeSoftware-1.2.3.tar.gz
mkdir build
cd build/
cmake -DCMAKE_INSTALL_PREFIX=/usr/local/SomeSoftware/1.2.3 ../SomeSoftware-1.2.3/
make
[sudo] make install
```

## Separate Source and Build Directories

As you could see from the previous example already, the vast majority of CMake based projects can be "built out of source". Such that the build products are kept completely separately from the source files/directories. It's worth to point out however that different projects may have different preferences for being built in- or out-of-source.

To execute a default in-source build, you'd do the following:

```
tar -xvzf SomeSoftware-1.2.3.tar.gz
cd SomeSoftware-1.2.3
cmake
make
```

In ATLAS we only officially support out-of-source builds of the software at the moment. In-source builds should in principle work, but the clear recommendation is that you should always use out-of-source builds with the ATLAS code.

## CMake 101

In this section we show you some completely standalone CMake examples. To get you familiar with just the very basics of the CMake configuration file syntax.

### Hello World!

---

As always, let's start with a Hello World example. Set up a directory like `tutorial/helloworld/source` for this. And then write a simple `cxx` file in it. Let's call the source file `helloworld.cxx`.

Building this source file is of course extremely simple. You can just call `g++ -o helloworld helloworld.cxx` or `clang++ -o helloworld helloworld.cxx`. You can just try doing this as a first step.

Now, let's write a configuration for CMake that does practically the same. Every CMake project must have a file called `CMakeLists.txt` in its root directory. In this case let's put a file into the same directory (`tutorial/helloworld/source`) as the source file. (Strictly speaking it's not mandatory to put the project's main `CMakeLists.txt` file in the root directory of the project. But it's a widely accepted convention to organise code like this.)

The first (technical) thing that every `CMakeLists.txt` file has to declare is a minimum CMake version that's required to interpret it. Expressed like:

```
cmake_minimum_required( VERSION 2.8 )
```

Next, one has to declare a name for the project that we're configuring/building. This is done like:

```
project( HelloWorld )
```

You always have to include these two statements in your configuration file.

Next, let's declare to CMake that it should build an executable. This can be done with the [add\\_executable](#) function. Like this:

```
add_executable( helloworld helloworld.cxx )
```

Now, create a directory called `tutorial/helloworld/build`, and execute:

```
cd tutorial/helloworld/build/  
cmake ../source  
make
```

You should see output like:

You should now have an executable called `helloworld` in your build directory. Go ahead, and try running it.

### Library and Executable

---

Let's now start building something a bit more complicated. A small project that builds a shared or static library, and an executable that makes use of this library. Let's not make the source of it too complicated. You should just create 3 simple source files in a directory layout like:

- `tutorial/fullexample/source/lib/mylibrary.h`
- `tutorial/fullexample/source/lib/mylibrary.cxx`
- `tutorial/fullexample/source/app/myexecutable.cxx`

You could write your example code for this if you like, or just take the simple implementation from here.

Now let's try to build this project...

### Building the Library

---

We start out with a `tutorial/fullexample/source/CMakeLists.txt` file like in the first example. Holding just:

```
# Mandatory setting for minimal CMake requirement:
cmake_minimum_required( VERSION 2.8 )
```

```
# Create a project:
project( FullExample )
```

Building a library can be done using the [add\\_library](#) function. It has a syntax very similar to [add\\_executable](#). To build a default **static** library, you can just write:

```
add_library( MyLibrary lib/mylibrary.cxx )
```

Note that you only have to specify the source file that makes up your library. **But** it's a very good habit to list both the header and source files that make up a component in the component's declaration. As in this case when CMake generates an IDE project for something like [Xcode](#) or [VisualStudio](#), the header files would show up correctly in the IDE. So in this case you should preferably write:

```
add_library( MyLibrary lib/mylibrary.h lib/mylibrary.cxx )
```

Try building your project. It should produce a static library called `libMyLibrary.a` in your build directory. Now, you should try building a shared library instead. To do this, you just need to modify your library declaration to look like:

```
add_library( MyLibrary SHARED lib/mylibrary.h lib/mylibrary.cxx )
```

If you build the project now, you should get a library called `libMyLibrary.so` (or `libMyLibrary.dylib` on MacOS X) generated in your build directory.

**Note:** With recent versions of CMake on [MacOS X](#) you will get a warning during the configuration of the build of the shared library. You can silence that warning by adding the following line in your configuration before the `add_library` line:

```
set( CMAKE_MACOSX_RPATH ON )
```

## Building the Executable

As a first iteration let's just try to build the executable like we did in the Hello World example. By just using:

```
add_executable( MyExecutable app/myexecutable.cxx )
```

As you should expect, this will not just work like this. If you try building your project with a configuration like this, you should see something like the following:

```
Scanning dependencies of target MyExecutable
[ 25%] Building CXX object CMakeFiles/MyExecutable.dir/app/myexecutable.cxx.o
/Users/krasznai/Development/cmake/tutorial/fullexample/source/app/myexecutable.cxx:3:10: fatal error: 'mylibrary.h' file
      not found
#include "mylibrary.h"
      ^
1 error generated.
make[2]: *** [CMakeFiles/MyExecutable.dir/app/myexecutable.cxx.o] Error 1
make[1]: *** [CMakeFiles/MyExecutable.dir/all] Error 2
make: *** [all] Error 2
```

Let's work around this issue for now by modifying the `myexecutable.cxx` source file to include the header of the library with the relative `"../lib/mylibrary.h"` path instead. Once you do that, the build will fail like:

```
Scanning dependencies of target MyExecutable
[ 25%] Building CXX object CMakeFiles/MyExecutable.dir/app/myexecutable.cxx.o
[ 50%] Linking CXX executable MyExecutable
Undefined symbols for architecture x86_64:
  "simpleFunction()", referenced from:
      _main in myexecutable.cxx.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
make[2]: *** [MyExecutable] Error 1
make[1]: *** [CMakeFiles/MyExecutable.dir/all] Error 2
make: *** [all] Error 2
```

The problem here is of course that we never declared that the executable has to use the library that we are also building in the project. We do this by using the [target\\_link\\_libraries](#) command. By putting the following line after the declaration of the executable and library:

```
target_link_libraries( MyExecutable MyLibrary )
```

With this added, the project should finally build both the library and the executable.

## Component Properties

Let's now come back to the first compilation problem that we bumped into. That the `myexecutable.cxx` source file tried to include the library's header simply with "mylibrary.h". Which would be very desirable, as that source file should not have to know where that header is exactly in the source tree. Or if it's even inside of the same source tree, or in some external location. So let's change the include statement back to `#include "mylibrary.h"` and try to make the build work.

The blunt approach for this may be to simply use the [include\\_directories](#) function. This function allows us to declare additional include paths for all of the components in the project. You may just add the following to the configuration to make it work:

```
include_directories( ${CMAKE_SOURCE_DIR}/lib )
```

Note the usage of `${CMAKE_SOURCE_DIR}` variable in this expression. At this point it should be mentioned how CMake deals with variables in the configuration files. You can set a variable using the [set](#) function. After setting a variable, you can refer to it in consecutive lines with the `${varname}` formalism.

Now, using [include\\_directories](#) will make this tiny project work correctly, but it's not a technique that would work nicely for large projects. We don't want to declare all components to use this include path all the time. We just want components that need to link against the `MyLibrary` library to use this include path. CMake has a system of doing this. Using the [target\\_include\\_directories](#) function. Now in this case, the logical setup is that everything that links against `MyLibrary` should be seeing files under `lib/`. We can do this by adding the following to the configuration after declaring the library:

```
target_include_directories( MyLibrary PUBLIC ${CMAKE_SOURCE_DIR}/lib )
```

After adding this, just by declaring that `MyExecutable` needs to link against `MyLibrary`, it will receive this include path automatically for its build. Also, if you would set up the build of another library, that needs to link against `MyLibrary`, that new library, and all of its clients, would see `lib/` in their include paths.

## Installing Targets and Files

As we discussed at the start, CMake follows the configure - build - install paradigm. So far we only discussed the first two. In order to provide the library and the executable to third parties, you need to declare how to install them. This is done using the [install](#) function. This is a fairly rich function, which can describe the installation of a **lot** of different things. We don't try to describe all of its features here.

Let's just imagine that we want to install the executable under some `prefix/bin` and the library under some `prefix/lib` location. As is usual on POSIX platforms. We can do this by adding the following to the configuration:

```
install( TARGETS MyLibrary MyExecutable
  RUNTIME DESTINATION bin
  LIBRARY DESTINATION lib )
```

By default CMake will try to install things under `/usr/local` on POSIX platforms. So if you just try to call `make install` in your project, it will try to install the executable as `/usr/local/bin/MyExecutable`, and the library as `/usr/local/lib/libMyLibrary.so`. For testing purposes we usually don't want to install things under `/usr/local`. To tell the build to install the files into some more "innocent" location, let's remember what was shown at the start of this tutorial. And build the project like:

```
cmake -DCMAKE_INSTALL_PREFIX=../install ../source
make
make install
```

You should see output like:

```
[bash][tauriel]:build > make
Scanning dependencies of target MyLibrary
[ 25%] Building CXX object CMakeFiles/MyLibrary.dir/lib/mylibrary.cxx.o
[ 50%] Linking CXX shared library libMyLibrary.dylib
[ 50%] Built target MyLibrary
Scanning dependencies of target MyExecutable
[ 75%] Building CXX object CMakeFiles/MyExecutable.dir/app/myexecutable.cxx.o
[100%] Linking CXX executable MyExecutable
[100%] Built target MyExecutable
[bash][tauriel]:build > make install
[ 50%] Built target MyLibrary
[100%] Built target MyExecutable
Install the project...
-- Install configuration: ""
-- Installing: /Users/krasznai/Development/cmake/tutorial/fullexample/install/lib/libMyLibrary.dylib
-- Installing: /Users/krasznai/Development/cmake/tutorial/fullexample/install/bin/MyExecutable
```

This is pretty good already. But notice that we only installed the library. A third party would also need `mylibrary.h` to be able to use it. To install this header file in the POSIX standard location, add the following to the project configuration:

```
install( FILES lib/mylibrary.h DESTINATION include )
```

Check for yourself how this will install the header. At this point you should have a configuration file like the one shown here.

This concludes the initial part of the CMake tutorial.

## Semi-advanced CMake

In this part we show you some techniques needed to build projects that go beyond the absolute basics. This should help you understand how we build real ATLAS code later on.

### Finding Externals

CMake's main function for finding pieces of software that are not built as part of the current project, is [find\\_package](#). CMake out of the box is able to find the headers/libraries of a large set of externals. For instance, if you want to use Boost in your code, you'd do:

```
find_package( Boost )
```

Technically what happens in this case is that CMake calls on a file called [FindBoost.cmake](#), which takes care of locating Boost, and then setting a number of variables that you can make use of in your CMake configuration.

In general every `FindFoo.cmake` module is allowed to set variables with any name. It is by convention that almost all of them set the following variables:

- `FOO_FOUND`: A boolean variable that you can use to determine if the external was found.
- `FOO_INCLUDE_DIRS`: A list of include directories necessary to build against this external.
- `FOO_LIBRARIES`: A list of libraries that one needs to link against to use this external.

Now, let's try this out. Add some code to `myexecutable.cxx` that relies on Boost. If you need some inspiration, you can take my example.

In order to build my version of `myexecutable.cxx`, you could use the following configuration:

```
find_package( Boost COMPONENTS program_options )
...
target_include_directories( MyExecutable PRIVATE ${Boost_INCLUDE_DIRS} )
target_link_libraries( MyExecutable ${Boost_LIBRARIES} )
```

Note the extra option I gave to the `find_package` call. Find modules tend to be different one-by-one on this. But most of them follow the rule that if they point to a toolset that is composed of multiple libraries, then you need to specify using `COMPONENTS` which libraries you'll want to use from the external.

Also note that you can call [target\\_include\\_directories](#) and [target\\_link\\_libraries](#) on the same component multiple times. They will just append to the already existing configuration, a new function call will not erase what was set up before. But you can also try to merge these two calls into the previous ones, by checking the online descriptions of the functions how exactly to do this.

If Boost is available on your system in some default location, your build should succeed with these changes. If you have Boost in some non-conventional location, like under `/usr/local/boost/1.59.0`, then you need to refer to [FindBoost.cmake](#) to see how to give hints to CMake about the location of Boost. In this case you'd need to configure the build like:

```
cmake -DBOOST_ROOT=/usr/local/boost/1.59.0/ ../source/
```

## A Note on ROOT libraries

You may find that you get errors from the linking step of the build complaining about undefined references. For example

```
undefined reference to `TCanvas::TCanvas(char const*, char const*, int)'
```

This (often) means that the correct libraries aren't being linked by CMake. For an external package (one for which you have to use `find_package`) this might mean that the correct library is not being included in the list that `find_package` has retrieved (it might also mean that you haven't added `$(ROOT_LIBRARIES)` to `LINK_LIBRARIES` ...). In order to tell CMake which libraries to find you add to the `COMPONENTS` list in the `find_package` call; for instance to fix the linker error shown above you would add `Gpad` to this list. This call might then look like

```
find_package( ROOT COMPONENTS MathCore RIO Core Tree Hist Gpad )
```

To find the correct library you need look at the relevant page in the ROOT documentation (search for libraries). For example on the [TCanvas](#) page there is a graphic at the bottom. The entry in the grey box (`libGpad`) is the library name - remove the 'lib' from the front and you have the correct component name!

## Exporting Your Project

The situation with [find\\_package](#) is actually even a bit more complicated than shown so far. When you call `find_package(Foo)` in your configuration, CMake tries a number of different things to figure out what you meant. Making use of a file called `FindFoo.cmake` is one of the last steps.

The preferred solution is that CMake looks for a file called `FooConfig.cmake` (some other spellings are also possible) that does not look for the project in question. But rather describes exactly where a given project is installed. For instance, if you look under

```
/afs/cern.ch/sw/lcg/releases/ROOT/6.06.06-a2c9d/x86_64-slc6-gcc49-opt/cmake
```

, you'll find a file called `ROOTConfig.cmake`. Since CMake doesn't provide a `FindROOT.cmake` module, by default if you need to use ROOT in a standalone application, you need to rely on this configuration that comes with the ROOT installation. Note that if you use ROOT's `thisroot.csh` file for setting it up in your environment, you should be able to call `find_package(ROOT)` to find it in your CMake project.

So, to make the results of your build visible to downstream CMake projects, that are independent from your project, you need to provide such a configuration file with your installation. This is done with [install](#)'s "export feature". To do this, you first of all need to label which of your installed targets you want to make visible in this CMake configuration. Using the `EXPORT` argument of `install`, like:

```
install( TARGETS MyLibrary MyExecutable
  EXPORT MyProjectConfig
  RUNTIME DESTINATION bin
  LIBRARY DESTINATION lib )
```

And then, once you declared all your targets for export that you wanted to, you create the configuration file like:

```
install( EXPORT MyProjectConfig
  DESTINATION cmake )
```

Note that the name of your export should end in "Config". Unless you want to use an even more elaborate exporting scheme, where you provide the "config file" yourself, which makes use of the file created by CMake internally. (This is what the ATLAS project configuration does...)

To be able to find your freshly exported project in some downstream project, one can use the `CMAKE_PREFIX_PATH` environment variable. As the `find_package` function parses all paths in this environment variable to look for the project that you asked of it.

## Making Use of This Project in a Second One

**Note:** This exercise will only work correctly after you followed the instructions in [Generator Expressions](#).

To demonstrate how you can use the project that you just now installed into a directory on your disk, let's create a second project from scratch. Put it into a new directory, something like `tutorial/secondproject/source`. Let's just use a very simple executable that includes the header file implemented in the first project, and then makes use of the first project's library. This could look like:

```
// Include the header from the first project:
#include "mylibrary.h"

int main() {

    // Call the library's function twice:
    simpleFunction();
    simpleFunction();

    // Return gracefully:
    return 0;
}
```

I just called this file `otherexecutable.cxx` myself. Now, to build this source file, let's use the following configuration:

```
# Set up the project:
cmake_minimum_required( VERSION 2.8 )
project( SecondProject )

# Find the first project:
find_package( MyProject )

# Set up the executable of this project:
add_executable( OtherExecutable otherexecutable.cxx )

# Link it against MyLibrary from the first project:
target_link_libraries( OtherExecutable MyLibrary )
```

As noted before, you need to set the `CMAKE_PREFIX_PATH` environment variable for this to work correctly. To configure the build, you'll need to do something like:



```
cd secondproject/
mkdir build
cd build/
export CMAKE_PREFIX_PATH=/Users/krasznaa/Development/cmake/tutorial/fullexample/install
cmake ../source
```

Notice that neither the second project's source or configuration had any hardcoded information about the first project's location. Also note that in this case we didn't make use of variables like `${MyProject_INCLUDE_DIRS}` and `${MyProject_LIBRARIES}`. Instead, we used the first project's `MyLibrary` target as if it were configured in this second project. In case you're interested in how this works, read up about "imported libraries" in the [add\\_library](#) configuration.

## Generator Expressions

If you did everything correctly so far, in the previous step you should've seen the following error during CMake configuration:

CMake Error in CMakeLists.txt:

Target "MyLibrary" INTERFACE\_INCLUDE\_DIRECTORIES property contains path:

```
"/Users/krasznaa/Development/cmake/tutorial/fullexample/source/lib"
```

which is prefixed in the source directory.

This did not prevent you from building and installing your project, but it did highlight a serious mistake in our configuration.

When CMake exports the description of a library, it captures all knowledge about how to link client code against it. What include directories the client needs to use it, what other libraries the client may need to link against, etc. We declared previously that in order to use the library, one would need to include the `lib/` sub-directory of the source tree. With the line:

```
target_include_directories( MyLibrary PUBLIC ${CMAKE_SOURCE_DIR}/lib )
```

But this path of course has no meaning once the project is installed. Especially since the header describing the library is under `include/` in the installation tree. It is at this point that CMake [generator expressions](#) come to the rescue. This problem can be resolved with the most often used generator expression pattern, like:

```
target_include_directories( MyLibrary PUBLIC
  $<BUILD_INTERFACE:${CMAKE_SOURCE_DIR}/lib>
  $<INSTALL_INTERFACE:include> )
```

Generator expressions can be used for configuring fairly complex build setups. So it's a good idea to read about them more. But in this tutorial we can't spend any more time to look at them.

## Configuration Control Flow

As you may have concluded it by now, the CMake configuration files behave very much like executable scripts. Where you call functions, that do various tasks. In fact, it is possible to insert a fair bit of logic into CMake configurations. With functions such as [if](#), [while](#), [foreach](#), [break](#), and lot's of others, one can write code like:

```
# Source files that have no dependencies on externals:
set( source_files src/source1.cxx src/source2.cxx )

# Look for Boost:
find_package( Boost )

# If Boost is found, build some extra source files as well, which depend on it:
if( Boost_FOUND )
    message( STATUS "Boost found, adding Boost-specific sources to the build" )
    list( APPEND source_files src/boost_source1.cxx src/boost_source2.cxx )
endif()

# We don't support Windows... :-P
if( WIN32 )
    message( WARNING "Compilation on Windows platform is not supported. It will likely fail." )
endif()

# Set platform specific build options:
if( APPLE )
    set( CMAKE_MACOSX_RPATH OFF )
endif()
```

## Functions and Macros

---

CMake allows you to seamlessly integrate new function calls into your configuration. Allowing these function calls to behave just like the built in ones. CMake actually provides two ways of implementing function-like constructs:

- [function](#) behaves as a C/C++ function would. It receives arguments from the caller, but is executed in an environment separated from that of the caller. So any variable created inside the function body gets deleted at the end of it, not polluting the caller's environment.
- [macro](#) behaves as a C/C++ pre-processor macro would. The code of the macro is executed in the environment of the caller. And every new variable created by the macro remains visible in the caller's environment after the return of the macro.

The way to write functions and macros properly in CMake configuration files is too large of a topic to handle properly in this tutorial. Here I'd just show some extremely simple examples.

```
function( testFunction arg1 arg2 )
    message( STATUS "Received arguments ${arg1} and ${arg2}" )
endfunction( testFunction )

testFunction( "Foo" "Bar" )

macro( project_add_executable name )
    add_executable( ${name} src/${name}.cxx )
    target_link_libraries( ${name} PRIVATE ${project_libraries} )
endmacro( project_add_executable )

foreach( exec writeFile readFile modifyFile )
    project_add_executable( ${exec} )
endforeach()
```

## Subdirectories

---

So far we have written a single `CMakeLists.txt` file for our project. For small projects this is usually enough. But for larger projects people usually want to split the build configuration into smaller chunks. CMake allows you to do this using the [add\\_subdirectory](#) function.

Let's try this out by adding a separate `CMakeLists.txt` file to the `app/` and `lib/` subdirectory of our project. The `CMakeLists.txt` files in subdirectories are in general not fully functional files that could be given to a `cmake` command directly. Although it is possible to configure a project in such a way as well. So, unlike the main project file, that has to have a call to [cmake\\_minimum\\_required](#) and [project](#) in them, these configuration files can be written pretty much in any way.

In our example we just cut and paste the parts describing the build of the library and the executable into these directory specific files. Making `app/CMakeLists.txt` file contain:

```
# Find Boost:
find_package( Boost COMPONENTS program_options )

# Build the executable:
add_executable( MyExecutable myexecutable.cxx )
target_link_libraries( MyExecutable MyLibrary )
target_include_directories( MyExecutable PRIVATE ${Boost_INCLUDE_DIRS} )
target_link_libraries( MyExecutable ${Boost_LIBRARIES} )

# Install the executable:
install( TARGETS MyExecutable
        EXPORT MyProjectConfig
        RUNTIME DESTINATION bin )
```

And `lib/CMakeLists.txt` should contain:

```
# Build the library:
add_library( MyLibrary SHARED mylibrary.h mylibrary.cxx )
target_include_directories( MyLibrary PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}>
    $<INSTALL_INTERFACE:include> )

# Install the library:
install( TARGETS MyLibrary
        EXPORT MyProjectConfig
        LIBRARY DESTINATION lib )

# Install the header:
install( FILES mylibrary.h
        DESTINATION include )
```

Note that I now used file paths relative to the sub-directories that the `CMakeLists.txt` files are in, and how I used `CMAKE_CURRENT_SOURCE_DIR` in the code. At this point the `CMakeLists.txt` file in the root directory can look like:

```
# Mandatory setting for minimal CMake requirement:
cmake_minimum_required( VERSION 2.8 )

# Create a project:
project( FullExample )

# Only necessary on MacOS X to silence a warning:
set( CMAKE_MACOSX_RPATH ON )

# Add the subdirectories:
add_subdirectory( lib )
add_subdirectory( app )

# Install a CMake description of this project:
install( EXPORT MyProjectConfig
        DESTINATION cmake )
```

Try building the code, and see if you still get the same results as before. Notice what happened to the `MyExecutable` executable and `libMyLibrary.so` library inside your build directory. How they are now placed inside the `app/` and `lib/` subdirectories.

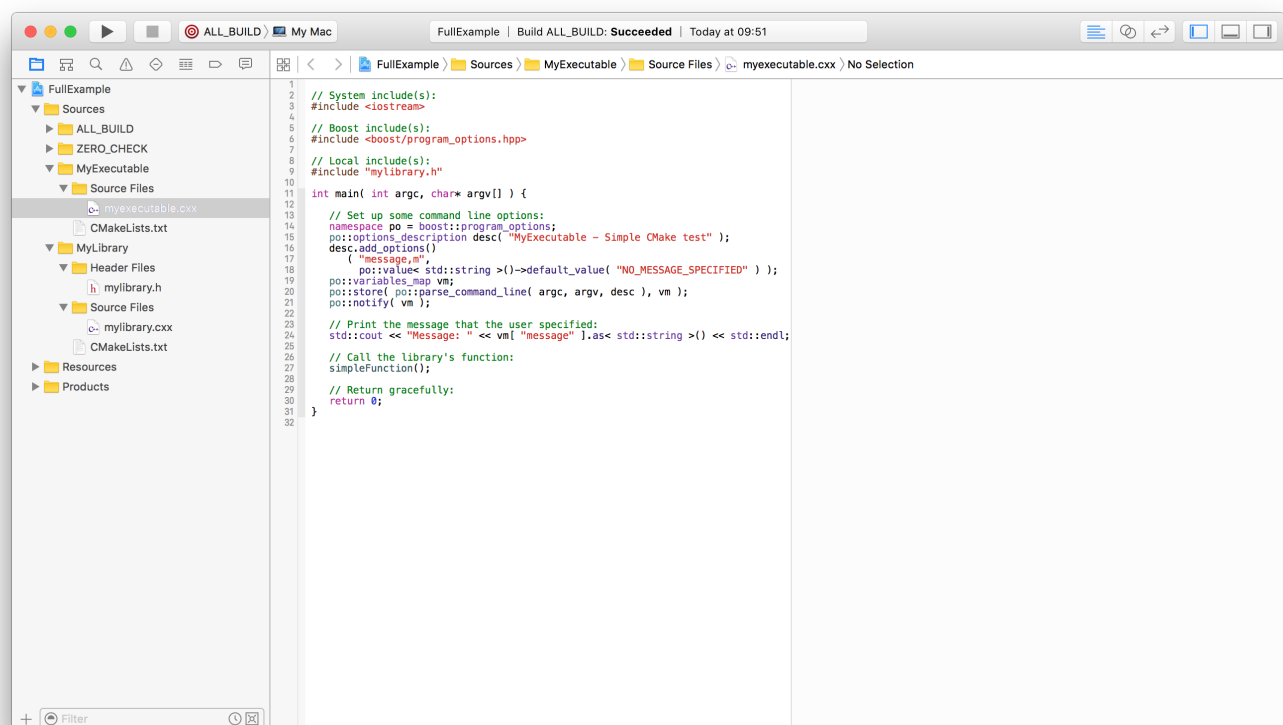
Finally, try flipping the order of including the two subdirectories. While logically building the library comes before building the application, this is not something that the main `CMakeLists.txt` file has to know. As long as the subdirectories declare their targets correctly, they can be parsed in any order, and CMake will figure out the correct way of building the sources.

## IDEs

As described in the beginning, CMake can generate configurations for many different build systems, not just GNU Make. For those of you running this exercise on MacOS X, you can generate an Xcode project from your sources with:

```
cmake -G Xcode ../source
open FullExample.xcodeproj
```

This should give you a project like:



Note that the source files end up in slightly awkward locations in the file browser. This can be improved by enabling the [USE\\_FOLDERS](#) global property, setting [FOLDER](#) properties on your targets, and possibly even using [source\\_group](#) to set the layout of the source files inside the target. But this is something that you should experiment with for yourself. The ATLAS project configuration does much of this for you behind the scenes already.

## CMake in ATLAS

In this part we show you how we build real ATLAS code.

When building the ATLAS offline and analysis projects, we make use of all of the features described so far. And then some... Every analysis/offline project is defined as a CMake project. So for instance the build of `AtlasReconstruction` is configured with a single `cmake` command, built with a single `make` command, and then installed with `make install`.

When we set up the build of `AtlasTrigger`, its configuration looks up the already installed `AtlasReconstruction` project using `find_package(AtlasReconstruction)`, and makes use of the targets imported from there to build its own targets.

The rest of this exercise will only work on CVMFS-capable SLC6 platforms. For the tutorial you should use lxplus.

## Setting Up the Environment

To set up the build environment on top of an offline/analysis project, we use [AtlasSetup](#). Create a work directory with the following layout:

```
tutorial/workdir/source
tutorial/workdir/build
```

Go to the `tutorial/workdir/source` directory, and do:

```
cd tutorial/workdir/source/
setupATLAS
asetup AtlasDerivation,21.0.19.1,here
```

If you now check the source directory, you'll see that `asetup` put a `CMakeLists.txt` file there.

Let's look at this file a bit. Your test area is set up as a separate project with CMake, named `WorkDir`. As you can see, the configuration looks up `AtlasDerivation`, so that it could build local packages against that project. The `atlas_project` function is responsible for setting up the build of the packages in your source directory. When you execute the CMake configuration, this function looks up all of the packages that you checked out into your source directory, and makes sure that components defined in them would be built correctly.

## Checkout and Build

While still in the source directory, let's check out a package, to test its compilation.

```
svnco -t DerivationFrameworkTools
```

Just execute `svnco -h` to see all the available options of this script.

Now, go to your build directory, and let's build this one package on top of `AtlasDerivation-21.0.19.1`.

```
cd ../build
cmake ../source
make
```

You'll see that the CMake configuration printed a **lot** more lines than the examples that you used previously. Unfortunately it also takes a bit longer to configure the build than in those simple cases, when CMake needs to learn about thousands of libraries while generating the build rules of your packages.

Notice that we don't usually talk about installation for a work area. You can however absolutely install it. The formalism for this is slightly different than what we showed so far. To install the build products of your project, you need to run something like:

```
make install DESTDIR=../install
```

But since your work directory doesn't need to be exposed to other people, you can normally just use your build products from the build directory directly. You don't need to install your test area first. Still just for information, you may want to try running this command to see what the project looks like after installation. You will see that it's not so different than what we've seen with the hand-written project.

## Package Configuration Files

As you can see, the example package has a `CMakeLists.txt` file in it. This file has the exact same purpose as the `CMakeListst.txt` files that we wrote for the `app/` and `lib/` subdirectories in [Subdirectories](#). It is **not** a functional build configuration by itself. It only works as part of an ATLAS project. Like the one set up by [AtlasSetup](#) in [Checkout and Build](#).

When writing a package `CMakeLists.txt` file, you need to use functions implemented in the [AtlasCMake](#) package. You can find a reference documentation about all of the available functions under [SoftwareDevelopmentWorkBookCMakeInAtlas](#). Here we just point out the most important things to know about writing such a file.

## Package Name and Dependencies

Every ATLAS package's configuration needs to start with declaring the package's name. This is done using the [atlas\\_subdir](#) function. Simply like:

```
atlas_subdir( MyPackage )
```

Then you need to declare which other packages this package of yours depends on. This is directly analogous to how package dependencies were declared in CMT. Dependencies can be either public or private. The rules for declaring a dependency as public or private are exactly the same as they were with CMT. This declaration is done using [atlas\\_depends\\_on\\_subdirs](#). Like:

```
atlas_depends_on_subdirs(
  PUBLIC
  Control/AthenaKernel
  PRIVATE
  Control/CxxUtils
  Control/AthenaBaseComps )
```

The [atlas\\_subdir](#) function declares a target called `Package_PkgName` for you. So that you would be able to build all targets in a single package by calling `make Package_MyPackage` in your build area. This can be useful when having many packages in your test area.

## Building Libraries and Executables

Probably the most important part of any package is how it builds its source files. To build a shared library that other packages/components can link against, you use [atlas\\_add\\_library](#). See the linked documentation for further details. But in general, you would use this function like:

```
atlas_add_library( MyLibrary MyLibrary/*.h src/*.cxx
  PUBLIC_HEADERS MyLibrary
  INCLUDE_DIRS ${Boost_INCLUDE_DIRS}
  PRIVATE_INCLUDE_DIRS ${ROOT_INCLUDE_DIRS}
  LINK_LIBRARIES ${Boost_LIBRARIES} AthenaKernel
  PRIVATE_LINK_LIBRARIES ${ROOT_LIBRARIES} CxxUtils AthenaBaseComps )
```

Note that unlike for the built-in [add\\_library](#) function, you can use wildcarded expressions for pointing to your source files. This makes the writing of configuration files convenient when you have a lot of source files. But it has the downside that whenever you add or remove a file, you have to re-run the `cmake` command in your build directory. Otherwise the build configuration will not learn about the addition/removal of the file.

We also have a different class of libraries in the ATLAS code called "component libraries". These are libraries that implement some Gaudi/Athena components. Such libraries may not be linked against in other packages. They are only meant to be loaded by the framework as they are. Component libraries are declared by [atlas\\_add\\_component](#). The function looks **very** similar to [atlas\\_add\\_library](#). You can call it like:

```
atlas_add_component( MyComponentLibrary src/*.h src/*.cxx src/components/*.cxx
  LINK_LIBRARIES AthenaKernel AthenaBaseComps )
```

Executables are built with [atlas\\_add\\_executable](#). Like:

```
atlas_add_executable( MyExecutable app/myexecutable.cxx
  LINK_LIBRARIES MyLibrary )
```

## Installing Files

ATLAS uses some conventions on where to put jobOptions, python files, data files, etc. in the installed directory to make them visible in your runtime environment. To simplify the task of installing such files in the correct place, [AtlasCMake](#) provides a number of helper functions. Described in [this section](#). In general, you just use these functions like:

```
atlas_install_python_modules( python/*.py )
atlas_install_joboptions( share/*.py )
```

## Building and Running (Unit) Tests

We use CTest to run (unit) tests in the ATLAS code. You declare tests using the [atlas\\_add\\_test](#) function. This function has a **lot** of options, allowing you to declare tests in many different ways.

It allows you to build an executable from some source files that then needs to be executed to run the test, like:

```
atlas_add_test( MyCompiledTest
  SOURCES test/simpleTest.cxx
  LINK_LIBRARIES MyLibrary )
```

It also allows you to implement an executable (python, shell, etc.) script in your package that will be run as the test. This allows you to even run fully fledged Athena jobs as a test if necessary. You can execute a script as a test with:

```
atlas_add_test( MyScriptTest
  SCRIPT test/scriptedTest.sh )
```

Once you built your project, you can execute all tests defined in all of the packages, with `make test`.

## Runtime Environment

This is an area that CMake provides very little support with. Most CMake projects you install into some system-wide location, so you don't need to care about setting up any environment variables to use them after installation.

ATLAS projects are very different. In order to be able to use an offline project, we need to set a **lot** of environment variables. This is done using tricks that are a bit beyond the level of this tutorial. Suffice it to say that at the end of the configuration step we generate a file called `setup.sh` that sets up the runtime environment of the project.

When you called `asetup`, that actually just "sourced" the `setup.sh` file of AtlasDerivation-21.0.19.1 from CVMFS. (After setting just a few environment variables itself.) Most of the environment setup comes from the file that was generated during the build of AtlasDerivation.

Just by calling `asetup`, you only get the environment variables necessary to use the base project that you asked for. Once you built your test area on top of this base release, you'll want to set up your runtime environment such that libraries/executables/python modules/etc. that are in your test area, would override those from the base release. You do this by doing:

```
source build/x86_64-slc6-gcc49-opt/setup.sh
```

This is a **very** important step that's unfortunately easy to forget. If you forget to execute this command, none of your locally built code will be taken into account when you try to run an Athena job for instance.

## Resources


- [https://cmake.org/Wiki/CMake\\_Useful\\_Variables](https://cmake.org/Wiki/CMake_Useful_Variables)
- [CMakeTestProjectInstructions](#)
- [SoftwareDevelopmentWorkBookCMakeInAtlas](#)
- [CMTCMakeRosettaStone](#)

### Major updates:

-- [LouiseHeelan](#) - 2016-09-26

Responsible: [LouiseHeelan](#)

Last reviewed by: **Never reviewed**

I	Attachment	History	Action	Size	Date	Who	Comment
	<a href="#">Screen_Shot_2016-10-04_at_09.52.05.png</a>	r1	<a href="#">manage</a>	528.9 K	2016-10-04 - 10:37	<a href="#">AttilaKrasznahorkay</a>	

Topic revision: r14 - 2017-11-22 - [JonathanThomasBurr](#)

Copyright &© 2008-2019 by the contributing authors. All material on this collaboration platform is the property of the contributing authors.  
Ideas, requests, problems regarding TWiki? [Send feedback](#)

