**How to Make an Executable Program**

The code for the user examples in Geant4 is placed in the subdirectory examples of the main Geant4 source package. This directory is installed to the share/Geant4-G4VERSION/examples (where G4VERSION is the Geant4 version number) subdirectory under the installation prefix. In the following sections, a quick overview will be given on how to build a concrete example, “ExampleB1”, which is part of the Geant4 distribution, using CMake and the older, and now deprecated, Geant4Make system.

**Using CMake to Build Applications**

Geant4 installs a file named Geant4Config.cmake located in

+- CMAKE\_INSTALL\_PREFIX

+- lib/

+- Geant4-G4VERSION/

+- Geant4Config.cmake

which is designed for use with the CMake find\_package command. Building a Geant4 application using CMake therefore involves writing a CMakeLists.txt script using this and other CMake commands to locate Geant4 and describe the build of your client application. Whilst it requires a bit of effort to write the script, CMake provides a very friendly yet powerful tool, especially if you are working on multiple platforms. It is therefore the method we recommend for building Geant4 applications.

We’ll use Basic Example B1, which you may find in the Geant4 source directory under examples/basic/B1, to demonstrate the use of CMake to build a Geant4 application. You’ll find links to the latest CMake documentation for the commands used throughout, so please follow these for further information. The application sources and scripts are arranged in the following directory structure:

+- B1/

+- CMakeLists.txt

+- exampleB1.cc

+- include/

| ... headers.hh ...

+- src/

... sources.cc ...

Here, exampleB1.cc contains main() for the application, with include/ and src/ containing the implementation class headers and sources respectively. This arrangement of source files is not mandatory when building with CMake, apart from the location of the CMakeLists.txt file in the root directory of the application.

The text file CMakeLists.txt is the CMake script containing commands which describe how to build the exampleB1 application

# (1)

cmake\_minimum\_required(VERSION 2.6 FATAL\_ERROR)

project(B1)

# (2)

option(WITH\_GEANT4\_UIVIS "Build example with Geant4 UI and Vis drivers" ON)

if(WITH\_GEANT4\_UIVIS)

find\_package(Geant4 REQUIRED ui\_all vis\_all)

else()

find\_package(Geant4 REQUIRED)

endif()

# (3)

include(${Geant4\_USE\_FILE})

include\_directories(${PROJECT\_SOURCE\_DIR}/include)

# (4)

file(GLOB sources ${PROJECT\_SOURCE\_DIR}/src/\*.cc)

file(GLOB headers ${PROJECT\_SOURCE\_DIR}/include/\*.hh)

# (5)

add\_executable(exampleB1 exampleB1.cc ${sources} ${headers})

target\_link\_libraries(exampleB1 ${Geant4\_LIBRARIES})

# (6)

set(EXAMPLEB1\_SCRIPTS

exampleB1.in

exampleB1.out

init\_vis.mac

run1.mac

run2.mac

vis.mac

)

foreach(\_script ${EXAMPLEB1\_SCRIPTS})

configure\_file(

${PROJECT\_SOURCE\_DIR}/${\_script}

${PROJECT\_BINARY\_DIR}/${\_script}

COPYONLY

)

endforeach()

# (7)

install(TARGETS exampleB1 DESTINATION bin)

For clarity, the above listing has stripped out the main comments (CMake comments begin with a “#”) you’ll find in the actual file to highlight each distinct task:

1. Basic Configuration

The cmake\_minimum\_required command simply ensures we’re using a suitable version of CMake. Though the build of Geant4 itself requires CMake 3.3 and we recommend this version for your own projects, Geant4Config.cmake can support the 2.6 and 2.8 series. The project command sets the name of the project and enables and configures C and C++ compilers.

1. Find and Configure Geant4

The aforementioned find\_package command is used to locate and configure Geant4 (we’ll see how to specify the location later when we run CMake), the REQUIRED argument being supplied so that CMake will fail with an error if it cannot find Geant4. The option command specifies a boolean variable which defaults to ON, and which can be set when running CMake via a -D command line argument, or toggled in the CMake GUI interfaces. We wrap the calls to find\_package in a [conditional block](https://cmake.org/cmake/help/v3.3/command/if.html) on the option value. This allows us to configure the use of Geant4 UI and Visualization drivers by exampleB1 via the ui\_all vis\_all “component” arguments to find\_package. These components and their usage is described later.

1. Configure the Project to Use Geant4 and B1 Headers

To automatically configure the header path, and force setting of compiler flags and compiler definitions needed for compiling against Geant4, we use the include command to load a CMake script supplied by Geant4. The CMake variable named Geant4\_USE\_FILE is set to the path to this module when Geant4 is located by find\_package. We use the include\_directories command to add the B1 header directory to the compiler’s header search path. The CMake variable PROJECT\_SOURCE\_DIR points to the top level directory of the project and is set by the earlier call to the project command.

1. List the Sources to Build the Application

Use the globbing functionality of the file command to prepare lists of the B1 source and header files.

Note however that CMake globbing *is only used here as a convenience*. The expansion of the glob only happens when CMake is run, so if you later add or remove files, the generated build scripts will not know a change has taken place. Kitware strongly recommend listing sources explicitly as CMake automatically makes the build depend on the CMakeLists.txt file. This means that if you explicitly list the sources in CMakeLists.txt, any changes you make will be automatically picked when you rebuild. This is most useful when you are working on a project with sources under version control and multiple contributors.

1. Define and Link the Executable

The add\_executable command defines the build of an application, outputting an executable named by its first argument, with the sources following. Note that we add the headers to the list of sources so that they will appear in IDEs like Xcode.

After adding the executable, we use the target\_link\_libraries command to link it with the Geant4 libraries. The Geant4\_LIBRARIES variable is set by find\_package when Geant4 is located, and is a list of all the libraries needed to link against to use Geant4.

1. Copy any Runtime Scripts to the Build Directory

Because we want to support out of source builds so that we won’t mix CMake generated files with our actual sources, we copy any scripts used by the B1 application to the build directory. We use foreach to loop over the list of scripts we constructed, and configure\_file to perform the actual copy.

Here, the CMake variable PROJECT\_BINARY\_DIR is set by the earlier call to the project command and points to the directory where we run CMake to configure the build.

1. If Required, Install the Executable

Use the install command to create an install target that will install the executable to a bin directory under CMAKE\_INSTALL\_PREFIX.

If you don’t intend your application to be installable, i.e. you only want to use it locally when built, you can leave this out.

This sequence of commands is the most basic needed to compile and link an application with Geant4, and is easily extendable to more involved use cases such as platform specific configuration or using other third party packages (via find\_package).

With the CMake script in place, using it to build an application is a two step process. First CMake is run to generate buildscripts to describe the build. By default, these will be Makefiles on Unix platforms, and Visual Studio solutions on Windows, but you can generate scripts for [other tools like Xcode and Eclipse](https://cmake.org/cmake/help/v3.3/manual/cmake-generators.7) if you wish. Second, the buildscripts are run by the chosen build tool to compile and link the application.

A key concept with CMake is that we generate the buildscripts and run the build in a separate directory, the so-called *build directory*, from the directory in which the sources reside, the so-called *source directory*. This is the exact same technique we used when building Geant4 itself. Whilst this may seem awkward to begin with, it is a very useful technique to employ. It prevents mixing of CMake generated files with those of your application, and allows you to have multiple builds against a single source without having to clean up, reconfigure and rebuild.

We’ll illustrate this configure and build process on Linux/macOS using Makefiles, and on Windows using Visual Studio. The example script and Geant4’s Geant4Config.cmake script are vanilla CMake, so you should be able to use other Generators (such as Xcode and Eclipse) without issue.

**Building ExampleB1 with CMake on Unix with Makefiles**

We’ll assume, *for illustration only*, that you’ve copied the exampleB1 sources into a directory under your home area so that we have:

+- /home/you/B1/

+- CMakeLists.txt

+- exampleB1.cc

+- include/

+- src/

+- ...

Here, our *source directory* is /home/you/B1, in other words the directory holding the CMakeLists.txt file.

Let’s also assume that you have already installed Geant4 in your home area under, *for illustration only*, /home/you/geant4-install.

Our first step is to create a *build directory* in which build the example. We will create this alongside our B1 *source directory* as follows:

$ cd $HOME

$ mkdir B1-build

We now change to this *build directory* and run CMake to generate the Makefiles needed to build the B1 application. We pass CMake two arguments

$ cd $HOME/B1-build

$ cmake -DGeant4\_DIR=/home/you/geant4-install/lib64/Geant4-G4VERSION $HOME/B1

Here, the first argument points CMake to our install of Geant4. Specifically, it is the directory holding the Geant4Config.cmake file that Geant4 installs to help CMake find and use Geant4. You should of course adapt the value of this variable to the location of your actual Geant4 install. This provides the most specific way to point CMake to the Geant4 install you want to use. You may also use the CMAKE\_PREFIX\_PATH variable, e.g:

$ cd $HOME/B1-build

$ cmake -DCMAKE\_PREFIX\_PATH=/home/you/geant4-install $HOME/B1

This is most useful for system integrators as it may be extended via the environment or command line with paths to the install prefixes of additional required software packages.

The second argument to CMake is the path to the *source directory* of the application we want to build. Here it’s just the B1 directory as discussed earlier. You should of course adapt the value of that variable to where you copied the B1 source directory.

CMake will now run to configure the build and generate Makefiles and you will see output similar to

$ cmake -DGeant4\_DIR=/home/you/geant4-install/lib64/Geant4-G4VERSION $HOME/B1

-- The C compiler identification is GNU 4.9.2

-- The CXX compiler identification is GNU 4.9.2

-- Check for working C compiler: /usr/bin/gcc-4.9

-- Check for working C compiler: /usr/bin/gcc-4.9 -- works

-- Detecting C compiler ABI info

-- Detecting C compiler ABI info - done

-- Detecting C compile features

-- Detecting C compile features - done

-- Check for working CXX compiler: /usr/bin/g++-4.9

-- Check for working CXX compiler: /usr/bin/g++-4.9 -- works

-- Detecting CXX compiler ABI info

-- Detecting CXX compiler ABI info - done

-- Detecting CXX compile features

-- Detecting CXX compile features - done

-- Configuring done

-- Generating done

-- Build files have been written to: /home/you/B1-build

The exact output will depend on the UNIX variant and compiler, but the last three lines should be identical to within the exact path used.

If you now list the contents of you build directory, you can see the files generated:

$ ls

CMakeCache.txt exampleB1.in Makefile vis.mac

CMakeFiles exampleB1.out run1.mac

cmake\_install.cmake init\_vis.mac run2.mac

Note the Makefile and that all the scripts for running the exampleB1 application we’re about to build have been copied across. With the Makefile available, we can now build by simply running make:

$ make -jN

CMake generated Makefiles support parallel builds, so N can be set to the number of cores on your machine (e.g. on a dual core processor, you could set N to 2). When make runs, you should see the output:

$ make

Scanning dependencies of target exampleB1

[ 16%] Building CXX object CMakeFiles/exampleB1.dir/exampleB1.cc.o

[ 33%] Building CXX object CMakeFiles/exampleB1.dir/src/B1PrimaryGeneratorAction.cc.o

[ 50%] Building CXX object CMakeFiles/exampleB1.dir/src/B1EventAction.cc.o

[ 66%] Building CXX object CMakeFiles/exampleB1.dir/src/B1RunAction.cc.o

[ 83%] Building CXX object CMakeFiles/exampleB1.dir/src/B1DetectorConstruction.cc.o

[100%] Building CXX object CMakeFiles/exampleB1.dir/src/B1SteppingAction.cc.o

Linking CXX executable exampleB1

[100%] Built target exampleB1

CMake Unix Makefiles are quite terse, but you can make them more verbose by adding the VERBOSE argument to make:

$ make VERBOSE=1

If you now list the contents of your *build directory* you will see the exampleB1 application executable has been created::

$ ls

CMakeCache.txt exampleB1 init\_vis.mac run2.mac

CMakeFiles exampleB1.in Makefile vis.mac

cmake\_install.cmake exampleB1.out run1.mac

You can now run the application in place:

$ ./exampleB1

Available UI session types: [ GAG, tcsh, csh ]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Geant4 version Name: geant4-10-04 [MT] (08-December-2017)

<< in Multi-threaded mode >>

Copyright : Geant4 Collaboration

Reference : NIM A 506 (2003), 250-303

WWW : http://cern.ch/geant4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<<< Reference Physics List QBBC

Visualization Manager instantiating with verbosity "warnings (3)"...

Visualization Manager initialising...

Registering graphics systems...

Note that the exact output shown will depend on how both Geant4 and your application were configured. Further output and behaviour beyond the Registering graphics systems... line will depend on what UI and Visualization drivers your Geant4 install supports. If you recall the use of the ui\_all vis\_all in the find\_package command, this results in all available UI and Visualization drivers being activated in your application. If you didn’t want any UI or Visualization, you could rerun CMake in your build directory with arguments:

$ cmake -DWITH\_GEANT4\_UIVIS=OFF .

This would switch the option we set up to false, and result in find\_package not activating any UI or Visualization for the application. You can easily adapt this pattern to provide options for your application such as additional components or features.

Once the build is configured, you can edit code for the application in its *source directory*. You only need to rerun make in the corresponding *build directory* to pick up and compile the changes. However, note that due to the use of CMake globbing to create the source file list, if you add or remove files, you must remember to rerun CMake to pick up the changes. This is another reason why Kitware recommend listing the sources explicitly.

**Using Geant4Make to build Applications**

*Please note that this system is deprecated, meaning that it is no longer supported and may be removed in future releases without warning. You should migrate your application to be built using CMake via the* Geant4Config.cmake *script, or any other build tool of your choice, using the* geant4-config *program to query the relevant compiler/linker flags.*

Geant4Make is the Geant4 GNU Make toolchain formerly used to build the toolkit and applications. It is installed on UNIX systems (except for Cygwin) for backwards compatibility with the Geant4 Examples and your existing applications which use a GNUmakefile and the Geant4Make binmake.gmk file. The files for Geant4Make are installed under:

+- CMAKE\_INSTALL\_PREFIX/

+- share/

+- geant4make/

+- geant4make.sh

+- geant4make.csh

+- config/

+- binmake.gmk

+- ...

The system is designed to form a self-contained GNUMake system which is configured primarily by environment variables (though you may manually replace these with Make variables if you prefer). Building a Geant4 application using Geant4Make therefore involves configuring your environment followed by writing a GNUmakefile using the Geant4Make variables and GNUMake modules.

To configure your environment, simply source the relevant configuration script CMAKE\_INSTALL\_PREFIX/share/Geant4-G4VERSION/geant4make/geant4make.(c)sh for your shell. Whilst both scripts can be sourced interactively, if you are using the C shell and need to source the script inside another script, you must use the commands:

$ cd CMAKE\_INSTALL\_PREFIX/share/Geant4-G4VERSION/geant4make

$ source geant4make.csh

or alternatively

$ source CMAKE\_INSTALL\_PREFIX/share/Geant4-G4VERSION/geant4make/geant4make.csh \

CMAKE\_INSTALL\_PREFIX/share/Geant4-G4VERSION/geant4make

In both cases, you should replace CMAKE\_INSTALL\_PREFIX with the actual prefix you installed Geant4 under. Both of these commands work around a limitation in the C shell which prevents the script locating itself.

Please also note that due to limitations of Geant4Make, you *should not* rely on the environment variables it sets for paths into Geant4 itself. In particular, note that the G4INSTALL variable *is not equivalent to* CMAKE\_INSTALL\_PREFIX.

Once you have configured your environment, you can start building your application. Geant4Make enforces a specific organization and naming of your sources in order to simplify the build. We’ll use Basic Example B1, which you may find in the Geant4 source directory under examples/basic/B1, as the canonical example again. Here, the sources are arranged as follows:

+- B1/

+- GNUmakefile

+- exampleB1.cc

+- include/

... headers.hh ...

+- src/

... sources.cc ...

As before, exampleB1.cc contains main() for the application, with include/ and src/ containing the implementation class headers and sources respectively. You must organise your sources in this structure with these filename extensions to use Geant4Make as it will expect this structure when it tries to build the application.

With this structure in place, the GNUmakefile for exampleB1 is very simple:

name := exampleB1

G4TARGET := $(name)

G4EXLIB := true

.PHONY: all

all: lib bin

include $(G4INSTALL)/config/binmake.gmk

Here, name is set to the application to be built, and it must match the name of the file containing the main() program without the .cc extension. The rest of the variables are structural to prepare the build, and finally the core Geant4Make module is included. The G4INSTALL variable is set in the environment by the geant4make script to point to the root of the Geant4Make directory structure.

With this structure in place, simply run make to build your application:

$ make

If you need extra detail on the build, append CPPVERBOSE=1 to the make command to see a detailed log of the command executed.

The application executable will be output to $(G4WORKDIR)/bin/$(G4SYSTEM)/exampleB1, where $(G4SYSTEM) is the system and compiler combination you are running on, e.g. Linux-g++. By default, $(G4WORKDIR) is set by the geant4make scripts to $(HOME)/geant4\_workdir, and also prepends this directory to your PATH. You can therefore run the application directly once it’s built:

$ exampleB1

If you prefer to keep your application builds separate, then you can set G4WORKDIR in the GNUmakefile before including binmake.gmk. In this case you would have to run the executable by supplying the full path.

Further documentation of the usage of Geant4Make and syntax and extensions for the GNUMakefile is described in the FAQ and Appendices of the [Geant4 User’s Guide for Application Developers](http://cern.ch/geant4/UserDocumentation/UsersGuides/ForApplicationDeveloper/html/index.html).

Please note that the Geant4Make toolchain is provided purely for convenience and backwards compatibility. We encourage you to use and migrate your applications to the new CMake and geant4-config tools. Geant4Make is deprecated from Geant4 10.0.

© Copyright 2017, Geant4 Collaboration.