Scripting Of CTest

Last edited by **Kitware Robot** 1 year ago

Testing With CTest

In Testing With CTest, you learned how to use CTest to perfom simple testing. That page also describes how to use CTest to submit dashboards. The typical sequence of commands that dashboard contributor will do involves:

```
cd /location/of/binary/Directory
rm -rf BinaryDirectory
mkdir BinaryDirectory
cd BinaryDirectory
ccmake /location/of/source/Directory
ctest -D Nightly
```

This is done on a typical UNIX system. On Windows system, the contributor needs to do something similar using Windows command interpreter, using something like Cygwin, or using a scripting language such as Perl, Python, or Tcl. Unfortunately none of these solutions is portable.

CTest Scripting

CTest has a built-in scripting mode, which significantly simplifies generating dashboards. Here is example of this script:

```
SET (CTEST_SOURCE_DIRECTORY "C:/Hoffman/My Builds/CMake")
SET (CTEST_BINARY_DIRECTORY "C:/Hoffman/My Builds/CMakeVSNMake71")

SET (CTEST_CVS_COMMAND "C:/cygwin/bin/cvs.exe")
SET (CTEST_CVS_CHECKOUT  "${CTEST_CVS_COMMAND} -d:pserver:hoffman@www.cmake.org:/cvsroot/CM

# which ctest command to use for running the dashboard
SET (CTEST_COMMAND
  "C:/Program Files/CMake/bin/ctest.exe -D Nightly"
  )

# what cmake command to use for configuring this dashboard
SET (CTEST_CMAKE_COMMAND
  "C:/Program Files/CMake/bin/cmake.exe"
  )


####################################################################
# The values in this section are optional you can either
# have them or leave them commented out
####################################################################

# should ctest wipe the binary tree before running
SET (CTEST_START_WITH_EMPTY_BINARY_DIRECTORY TRUE)

# this is the initial cache to use for the binary tree, be careful to escape
# any quotes inside of this string if you use it
SET (CTEST_INITIAL_CACHE "
MAKECOMMAND:STRING=nmake -i
CMAKE_MAKE_PROGRAM:FILEPATH=nmake
CMAKE_GENERATOR:INTERNAL=NMake Makefiles
BUILDNAME:STRING=Win32-nmake71
SITE:STRING=VOGON.kitware
CVSCOMMAND:FILEPATH=C:/cygwin/bin/cvs.exe
")

# set any extra environment variables to use during the execution of the script here:
SET (CTEST_ENVIRONMENT
)
```

ctest using git

```
set(CTEST_SOURCE_DIRECTORY "$ENV{HOME}/workspace/tmp/dashboards/libssh/source")
set(CTEST_BINARY_DIRECTORY "$ENV{HOME}/workspace/tmp/dashboards/libssh/build")

set(CTEST_SITE "host.libssh.org")
set(CTEST_BUILD_NAME "linux-gcc-default")

set(CTEST_CMAKE_GENERATOR "Unix Makefiles")
set(CTEST_BUILD_CONFIGURATION "Profiling")
set(CTEST_BUILD_OPTIONS "-DWITH_SSH1=ON -WITH_SFTP=ON -DWITH_SERVER=ON -DWITH_ZLIB=ON -DWIT

set(WITH_MEMCHECK TRUE)
set(WITH_COVERAGE TRUE)

#######################################################################

ctest_empty_binary_directory(${CTEST_BINARY_DIRECTORY})

find_program(CTEST_GIT_COMMAND NAMES git)
find_program(CTEST_COVERAGE_COMMAND NAMES gcov)
find_program(CTEST_MEMORYCHECK_COMMAND NAMES valgrind)

set(CTEST_MEMORYCHECK_SUPPRESSIONS_FILE ${CTEST_SOURCE_DIRECTORY}/tests/valgrind.supp)

if(NOT EXISTS "${CTEST_SOURCE_DIRECTORY}")
  set(CTEST_CHECKOUT_COMMAND "${CTEST_GIT_COMMAND} clone git://git.libssh.org/projects/libs
endif()

set(CTEST_UPDATE_COMMAND "${CTEST_GIT_COMMAND}")

set(CTEST_CONFIGURE_COMMAND "${CMAKE_COMMAND} -DCMAKE_BUILD_TYPE:STRING=${CTEST_BUILD_CONFI
set(CTEST_CONFIGURE_COMMAND "${CTEST_CONFIGURE_COMMAND} -DWITH_TESTING:BOOL=ON ${CTEST_BUIL
set(CTEST_CONFIGURE_COMMAND "${CTEST_CONFIGURE_COMMAND} \"-G${CTEST_CMAKE_GENERATOR}\"")
set(CTEST_CONFIGURE_COMMAND "${CTEST_CONFIGURE_COMMAND} \"${CTEST_SOURCE_DIRECTORY}\"")

ctest_start("Nightly")
ctest_update()
ctest_configure()
ctest_build()
ctest_test()
if (WITH_COVERAGE AND CTEST_COVERAGE_COMMAND)
  ctest_coverage()
endif (WITH_COVERAGE AND CTEST_COVERAGE_COMMAND)
if (WITH_MEMCHECK AND CTEST_MEMORYCHECK_COMMAND)
  ctest_memcheck()
endif (WITH_MEMCHECK AND CTEST_MEMORYCHECK_COMMAND)
ctest_submit()
```

To use SVN, just change the *CVSCOMMAND* line by *SVNCOMMAND* and give the right path to the *svn.exe* program. Also, don't forget to change the *LC_MESSAGES* environment variable to *en_US* in your script or directly in your system. This will force the SVN output language to english so that CTest will be able to analyse the data from SVN and submit them correctly to Dart2.

```
SET( ENV{LC_MESSAGES}    "en_US" )
```

If you want also the compiler output to be ASCII only (and cope with CDash integration), simply use:

```
SET( ENV{LANG}    "C" )
```

This script defined all the information CTest needs to submit CMake dashboard on Windows XP computer using NMake build system. The syntax used is the same as in CMake. Also, most commands available in CMake are available here, except things that require generator, such as TRY_COMPILE, ADD_LIBRARY, etc.

The script can be run by executing:

```
ctest -S /path/to/script.cmake
```

or in this case:

```
c:/Program Files/CMake/bin/ctest -S c:/Hoffman/DashboardScripts/vogon_cmake_nmake.cmake
```

Main Settings

Every CTest script has to contain the source and binary directory:

```
SET (CTEST_SOURCE_DIRECTORY "$ENV{HOME}/Dashboards/My Testing/Insight")
SET (CTEST_BINARY_DIRECTORY "$ENV{HOME}/Dashboards/My Testing/Insight-bin")
```

The "$ENV{HOME}" gets replaced by the environment variable "HOME", which on most systems points to user's home directory.

Second thing we need is the command that perform initial configuration of the project. For example, if the project uses CMake, then the command would be something like:

```
SET (CTEST_CMAKE_COMMAND "/usr/local/bin/cmake")
```

That assumes that CMake is installed in /usr/local. Similarly to setting up CMake command, we need to setup the way CTest will be called. Let say in your case you run CTest using:

```
ctest -D Nightly
```

Then the command in the CTest script should be:

```
SET (CTEST_COMMAND "/usr/local/bin/ctest -D Nightly")
```

The final important piece we need is the initial cache that CMake uses to configure the project. This should be minimal set of settings that will allow the system to test certain aspect of the project. Example initial cache would be:

```
SET (CTEST_INITIAL_CACHE "
//Name of generator.
CMAKE_GENERATOR:INTERNAL=Visual Studio 7

//Name of the build
BUILDNAME:STRING=Win32-vs70

//Name of the computer/site where compile is being run
SITE:STRING=DASH2.kitware

//Build VTK with shared libraries.
BUILD_SHARED_LIBS:BOOL=ON

//Path to the CVS
CVSCOMMAND:FILEPATH=C:/cygwin/bin/cvs.exe

//ITK TCL and Python wrapping
ITK_CSWIG_TCL:BOOL=ON
ITK_CSWIG_PYTHON:BOOL=ON
")
```

This will set the generator to be "Visual Studio 7", it will use shared libraries and turn on TCL and Python wrapping. Also, in the cache we can set the SITE and BUILDNAME variables, which will be used to display the testing results on the dashboard.

If you do not use default generator (such as Unix Makefiles), make sure you specify one. On systems that have multimple generators, it is a good idea to specify one always, since otherwise CMake might pick the wrong one.

Available generators are:

Generator	Description
Unix Makefiles	Unix style makefiles that can be processed using most Make processors except nonconforming processors such as NMake and Borland Make.
Borland Makefiles	Makefiles that can be processed using Borland Make processor
NMake Makefiles	Makefiles that can be processed using NMake Make processor
Visual Studio 6	Visual studio 6 project files

Generator	Description
Visual Studio 7	Visual Studio 7 project files (not same as Visual Studio 7 .NET 2003
Visual Studio 7 .NET 2003	Visual Studio 7 .NET 2003 project files
Visual Studio 8 2005	Visual Studi 8 (aka Visual Studio 2005) project files.

This is it. The CTest script can now be tested using:

```
ctest -S /path/to/script/script.cmake -V
```

The -V argument tells CTest to be extra verbose when doing dashboard.

More Settings

The main settings will generate dashboard for most systems. There are however issues in certain setups.

Most dashboard contributors want to start dashboard and not touch those systems for months. The problem is that the binary directory of the project will grow larger and larger. Common practice therefore is to wipe out binary directory every time dashboard is run. We can achieve this in CTest script by setting: CTEST_START_WITH_EMPTY_BINARY_DIRECTORY:

```
SET (CTEST_START_WITH_EMPTY_BINARY_DIRECTORY TRUE)
```

Now CTest will wipe out the binary tree every time. Make sure that the source tree is not equal to binary tree.

Second useful option is setting of environment in which CTest runs. Let say you have Cygwin, Borland, Visual Studio, and Intel compiler on your Windows system and you do not want to have system's global PATH pointing to various directories (Borland and Cygwin use "make", which one should be used). Now you want to do Borland dashboard. You can set environment variables using CTEST_ENVIRONMENT:

```
SET (CTEST_ENVIRONMENT
  "PATH=c:/WINDOWS/system32\;c:/WINDOWS\;c:/Programs/Borland/Bcc55/Bin")
```

Note that CTEST_ENVIRONMENT can accept multiple environment variables. For example:

```
SET (CTEST_ENVIRONMENT
  "DISPLAY=:0"
  "GLIBCPP_FORCE_NEW=1"
  "CC=gcc-3.4"
  "CXX=g++-3.4"
)
```

Another useful thing to do is to perform extra update on external repository. For example, VTK dashboard need to update VTK data. This can be achieved using CTEST_EXTRA_UPDATES_*. This also requires the CTEST_CVS_COMMAND to be set. An example of extra update is:

```
SET (CTEST_CVS_COMMAND "/usr/bin/cvs")
SET (CTEST_EXTRA_UPDATES_1 "$ENV{HOME}/Dashboards/MyTests/VTKData" "-dAP")
```

There can be up to ten CTEST_EXTRA_UPDATES_* settings: CTEST_EXTRA_UPDATES_1, CTEST_EXTRA_UPDATES_2, ..., CTEST_EXTRA_UPDATES_10.

Extra update can also be used to update DartConfig.cmake to the latest version, to receive settings right away instead of waiting until next day. An example of those kind of settings is a nightly start time.

```
SET (CTEST_EXTRA_UPDATES_2 "${CTEST_SOURCE_DIRECTORY}" " -dAP DartConfig.cmake")
```

Advanced Settings

When performing complex dashboards setups, some more CMake/CTest knowledge is needed. This section will describe solutions to some common CTest problems.

Memory Checking

Doing memory checking using Valgrind or similar product can take significant amount of time. There are two improvements the dashboard contributor can do to make dashboard more useful. First one is to submit partial results. This can be achieved by breaking down the CTest run into individual runs:

```
SET (CTEST_COMMAND
  "/usr/local/bin/ctest -D NightlyStart -D NightlyUpdate -D NightlyConfigure -D NightlyBuil
  "/usr/local/bin/ctest -D NightlyMemCheck -D NightlySubmit"
  )
```

The first CTest command will perform standard Nightly dashboard. The last two will perform only Memory checking and submission. This the whole dashboard submission will be there except the memory checking. Once the memory checking is completed, it will be submitted.

The second thing is to add special flags to make the CTest only test subset of tests:

```
SET (CTEST_COMMAND
  ...
  "/usr/local/bin/ctest -D NightlyMemCheck -I 0,,17"
  "/usr/local/bin/ctest -D NightlySubmit"
  )
```

The "-I 0,,17" will tell CTest to only run memory checking of every 17th test.

Initial Checkout

Let say you do not want to require the source of the project to be available. This is useful if the dashboards are performed on the temporary drive that can get erased. The following setting will checkout the source tree of CMake out of CVS repository before the testing is performed:

```
SET (CTEST_CVS_CHECKOUT
  "${CTEST_CVS_COMMAND} -q -z3
   -d :pserver:anoncvs@www.cmake.org:/cvsroot/CMake
   co -d \"CMake\" -D yesterday CMake")
```

Another useful feature is to backup source tree before performing dashboard. This way if the build fails, you can restore the old tree. This is useful when you require the source tree to be in good condition. This option has to be used in combination with CTEST_CVS_CHECKOUT. To turn this feature on, add this to CTest script:

```
SET (CTEST_BACKUP_AND_RESTORE TRUE)
```

Continuous Builds (old Style)

When setting up Continuous dashboard, the CTest has to be run periodically to check if there were any changes in the source. When the change appears, the full dashboard is tested. This can be achieved by running CTest from system scheduler or some script, or by using Continuous feature of CTest script. First of all, make sure that CTEST_COMMAND is using Continuous model:

```
SET (CTEST_COMMAND
  "C:/Program Files/CMake20/bin/ctest.exe -D Continuous"
  )
```

Also, if the number of tests is high, you may want to only run subset of tests:

```
SET (CTEST_COMMAND
  "C:/Program Files/CMake20/bin/ctest.exe -D Continuous -I ,,3"
  )
```

Now set the continuous parameters:

```
SET (CTEST_CONTINUOUS_DURATION 600)
SET (CTEST_CONTINUOUS_MINIMUM_INTERVAL 10)
SET (CTEST_START_WITH_EMPTY_BINARY_DIRECTORY_ONCE 1)
```

The CTEST_CONTINUOUS_DURATION is the duration of Continuous testing in minutes. In this case it will run Continuous dashboards for 10 hours. If it starts at 8 AM, the last continuous will be done around 6 PM. The CTEST_CONTINUOUS_MINIMUM_INTERVAL is the minimum interval between Continuous checks. In this case it will perform check every 10 minutes. CTEST_START_WITH_EMPTY_BINARY_DIRECTORY_ONCE will remove binary tree once the first Continuous is run. This way Continuous tests will take less time, since it will only rebuild modified files.

Continuous Builds (new Style)

CTEST_CONTINUOUS_DURATION is not used in the new style scripts. Instead, you you should use something like this:

```
while (${CTEST_ELAPSED_TIME} LESS 36000)
 set (START_TIME ${CTEST_ELAPSED_TIME})
 ctest_start (Continuous)
 ctest_update (RETURN_VALUE count)
 if (count GREATER 0)
  ctest_configure()
  ....
 endif ()
 ctest_sleep( ${START_TIME} 300 ${CTEST_ELAPSED_TIME})
endwhile()
```

Available Variables

When running CTest scripts, there are several variables available that can simplify writing CTest scripts. These variable include:

Variable	Description
CTEST_SCRIPT_DIRECTORY	Location of directory in which the CTest script is
CTEST_SCRIPT_NAME	Name of the CTest script
CTEST_SCRIPT_ARG	The extra arguments passed to CTest -S. For example, when running: `ctest -S /home/scripts/mysystem.cmake,Experimental` then the CTEST_SCRIPT_ARG will be set to Experimental. Then this can be used inside the script: `SET(MODEL Nightly)` `IF(${CTEST_SCRIPT_ARG} MATCHES Experimental)` `SET(MODEL Experimental)` `ENDIF(${CTEST_SCRIPT_ARG} MATCHES Experimental)` `SET (CTEST_COMMAND "/usr/local/bin/ctest -D ${MODEL}")` Now we can specify Nighlty or Experimental without editing CTest script.

Setting Up Cron/Scheduler

On Unix/MacOSX

Most Unix and Unix like systems use **cron**. Cron is a daemon that runs programs at specified times as described in crontab. Example of crontab is:

```
1  22   *   *   * /usr/bin/ctest -S /dash/dash1.cmake -V > /dash/logs/tests.log 2>&1
```

The part of the line:

```
1  22   *   *   *
```

specifies when to run the scheduled task. The columns correspond to minutes, hours, days, months, day of the week. This gets translated to 10:01 PM every day, every month. Another example would be:

```
47 6    * * 7
```

which is 6:47 AM on sunday.

The end of the line:

```
> /dash/logs/tests.log 2>&1
```

is korn-shell (/bin/sh) syntax for sending all messages (standard output and standard error) to a file **/dash/logs/tests.log**. If you set different shell in your crontab, make sure to use apropriate syntax.

To enable crontab, use the following command:

```
crontab -e
```

On Windows / Cygwin / MinGW

The following images are generated on Windows XP Pro, but the concepts should be the same on all Windows systems. Make sure to enable password for the user, otherwise scheduled tasks will not work.

1. Open "Scheduled Tasks" from Control Panel.

Sc_task_scheduler

2. Select **Add Scheduled Task**

Sc_start_new_scheduled_task

3. Select **Next** to select command.

Sc_select_browse

4. Click **Browse...** and select **CTest.exe**.

Sc_specify_command

5. Click **Next** and select name and repetition date. Repetition date for Nightly dashboards should be **Daily**.

Sc_specified_name_and_repetition

6. Click **Next** and select time to start the dashboard.

Sc_set_time

7. Click **Next** and select **Open advanced properties...** to fine tune the scheduled task.

Sc_open_advanced_settings

8. Select **Next** and type password of the user.

Sc_type_password

9. Task is created. The *Advanced Properties* dialog should open.

Sc_task_created

10. In advanced properties, specify full command name. This is very important that you use double quotes in case you have space in your path.

```
"c:/Path to ctest/ctest.exe" -S c:/Path to Dashboard Scripts/dashboard_script.cmake -V
```

Sc_proper_command_line

11. Select *'Ok*, which will ask for password again.

Sc_finish_password

12. The new task should be created.

Sc_new_task_exists

Conclusion

CTest scripts can significantly simplify dashboard contribution. Using simple template script with slight modification, the whole cluster of various systems can submit dashboards.

This page was initially populated by conversion from its original location in another wiki.