CMake/Testing With CTest

From KitwarePublic
< CMake

Contents

Introduction

CTest is a testing tool distributed as a part of CMake. It can be used to automate updating (using CVS for example), configuring, building, testing, performing memory checking, performing coverage, and submitting results to a CDash (http://www.cdash.org) or Dart (http://public.kitware.com/Dart) dashboard system.

There are two basic modes of operation for CTest.

In the first mode, CMake is used to configure and build a project, using special commands in the CMakeLists.txt file to create tests. CTest can then be used to execute the tests, and optionally upload their results to a dashboard server. This is what is handled in this tutorial.

In the second mode, CTest runs a script (using the same syntax as CMakeLists.txt) to control the whole process of checking out / updating source code, configuring and building the project, and running the tests. This is handled in CMake Scripting Of CTest.

Simple Testing

CMake has support for adding tests to a project:

```
enable_testing()
```

This adds another build target, which is test for Makefile generators, or RUN_TESTS for integrated development environments (like Visual Studio).

From that point on, you can use the ADD_TEST command to add tests to the project:

```
add_test( testname Exename arg1 arg2 ... )
```

Or, in its longer form:

```
add_test(NAME <name> [CONFIGURATIONS [Debug|Release|...]]
      [WORKING_DIRECTORY dir]
      COMMAND <command> [arg1 [arg2 ...]])
```

Once you have built the project, you can execute all tests via

```
make test
```

with Makefile generators, or by rebuilding the RUN_TESTS target in your IDE. Internally this runs CTest to actually perform the testing; you could just as well execute

```
ctest
```

in the binary directory of your build.

In some projects you will want to set *_POSTFIX properties on executables that will be executed for testing, e.g. to make executables compiled with debug information distinguishable ("<exename>-debug"). Note that the shorthand version of add_test does *not* automatically append these postfixes to the commands it calls for the test target, i.e. your test will want to call "<exename>" but the executable is "<exename>-debug", resulting in an error message. Use the long version of the add_test() in this case, which adds the appropriate _POSTFIX to the command name.

For more information, check the CMake Documentation (http://www.cmake.org/HTML/Documentation.html) or run:

```
cmake --help-command enable_testing
cmake --help-command add_test
cmake --help-property "<CONFIG>_POSTFIX"
cmake --help-command set_property
```

Dashboards

The next step is to not only execute the tests, but to log their results and provide them in such a way that they could be reviewed easily.

The result of a test run, reformatted for easy review, is called a "dashboard". A dashboard can be submitted to a central server, like CDash:

- open.cdash.org (http://http://open.cdash.org/index.php) and
- my.cdash.org (http://my.cdash.org).

There are three types of dashboard submissions:

- Experimental means the current state of the project. An experimental submission can be performed at any time, usually interactively from the current working copy of a developer.
- Nightly is similar to experimental, except that the source tree will be set to the state it was in at a specific nightly time. This ensures that all "nightly" submissions correspond to the state of the project at the same point in time. "Nightly" builds are usually done automatically at a preset time of day.
- Continuous means that the source tree is updated to the latest revision, and a build / test cycle is performed only if any files were actually updated. Like "Nightly" builds, "Continuous" ones are usually done automatically and repeatedly in intervals.

Dashboard Preparation

To enable the creation and submission of dashboards, add the following to your CMakeLists.txt:

```
include( CTest )
```

This module will automatically call enable_testing() (see above), so you no longer have to do so in your CMake files. It will also add several new targets to your build.

```
* The three main targets:
** Experimental
** Nightly
** Continuous
* For each of the above, targets for the intermediate steps
** ...Start
** ...Update
** ...Configure
** ...Build
** ...Submit
```

```
** ...Test
** ...Coverage
** ...MemCheck
** ...Submit
```

The intermediate targets are created so you could submit partial test results, or inspect the results before submitting (or continuing with more time-consuming steps like MemCheck).

All this can be disabled by setting the option `BUILD_TESTING` (which is also added by the CTest module and enabled by default) to OFF / false.

The default settings of the module are to submit the dashboard to Kitware's (http://www.kitware.com) Public Dashboard (http://public.kitware.com/Public/Dashboard/MostRecentResults-Nightly/Dashboard.html), where you can register your project for free.

In order to submit to some other server, "CTestConfig.cmake" in the top level directory of your source, and set your own dashboard preferences. If you are using a CDash server, you can download a preconfigured file from the respective project page on that server ("Settings" / "Project", tab "Miscellaneous").

An example of a CTestConfig.cmake:

```
## This file should be placed in the root directory of your project.
## Then modify the CMakeLists.txt file in the root directory of your
## project to incorporate the testing dashboard.
## # The following are required to uses Dart and the Cdash dashboard
##   ENABLE_TESTING()
##   INCLUDE(CTest)
set(CTEST_PROJECT_NAME "MyProject")
set(CTEST_NIGHTLY_START_TIME "01:00:00 UTC")

set(CTEST_DROP_METHOD "http")
set(CTEST_DROP_SITE "open.cdash.org")
set(CTEST_DROP_LOCATION "/submit.php?project=MyProject")
set(CTEST_DROP_SITE_CDASH TRUE)
```

Dashboard Creation

Once you have the above in place, you can build one of the targets added by the CTest module you included. Internally, this calls the CTest command line client, which you could also call directly instead:

```
ctest -D Experimental
```

A detailed description of CTest options can be seen by running:

```
ctest --help
```

A list of the available targets is listed by calling:

```
ctest -D help
```

Converting Dart to CTest

CTest is actually a fully Dart (http://www.itk.org/Dart/HTML/Index.shtml) compatible client, and could submit to any compatible server.

To convert existing Dart Client invocations to CTest, find lines like:

```
tclsh /location/of/Dart/Source/Client/DashboardManager.tcl DartConfiguration.tcl \
  Nightly Start Update Configure Build Test Submit
```

Then convert them to CTest style:

```
ctest -D Nightly
```

Advanced CTest

CTest has several additional features that include:

1. FTP/HTTP/SCP/XMLRPC submission support
2. Run individual tests, subset of tests, exclude tests, etc.
3. Dynamic analysis using Valgrind or Purify
4. Customization of the testing by providing:
    - Custom build error/warning regular expressions
    - Ability to suppress some tests from being tested or memory checked and ability to run subset of tests
    - Ability to run commands before and after tests are run
5. Ability to run whole testing process described in a single script

Submission Of Tests

CTest currently supports four methods directly and any other indirectly. Direct methods are HTTP, FTP, SCP and XML-RPC. Both HTTP and FTP methods require extra trigger mechanism, while SCP method relies on the fact that files are on the right place. To set the appropriate submission method, set CTEST_DROP_METHOD variable in CTestConfig.cmake.

Example for HTTP submission would be:

```
SET (CTEST_DROP_METHOD http)
SET (CTEST_DROP_SITE "public.kitware.com")
SET (CTEST_DROP_LOCATION "/cgi-bin/HTTPUploadDartFile.cgi")
SET (CTEST_TRIGGER_SITE
   "http://${DROP_SITE}/cgi-bin/Submit-CMake-TestingResults.pl")
```

where *http://public.kitware.com/cgi-bin/HTTPUploadDartFile.cgi* is a submit script and *http://public.kitware.com/cgi-bin/Submit-CMake-TestingResults.pl* is a trigger script.

For FTP submission:

```
SET (CTEST_DROP_METHOD ftp)
SET (CTEST_DROP_SITE "public.kitware.com")
SET (CTEST_DROP_LOCATION "/incoming")
SET (CTEST_DROP_SITE_USER "ftpuser")
SET (CTEST_DROP_SITE_PASSWORD "public")
SET (CTEST_TRIGGER_SITE
   "http://${DROP_SITE}/cgi-bin/Submit-CMake-TestingResults.pl")
```

where */incoming* is a location on the FTP site *public.kitware.com* with user *ftpuser* and password *public*. The trigger scrip is the same as with the http submit.

For XML-RPC submission (Dart2):

```
SET (CTEST_DROP_METHOD xmlrpc)
SET (CTEST_DROP_SITE "www.na-mic.org:8081")
SET (CTEST_DROP_LOCATION "PublicDashboard")
```

where XML-RPC submission is on the server *www.na-mic.org* with the port *8081*. The project name is *PublicDashboard*. XML-RPC submission does not require the trigger script.

Running Individual Tests

CTest supports two different ways of specifying subset of tests to run.

The first way is to specify the regular expression using -R and -E. -R specifies tests to be included and -E specifies the tests to be removed. For example, when running ctest in show-only mode, where no tests are run, we may see something like:

```
Test project
1/ 13 Testing PythonDataDesc
```

```
 2/ 13 Testing VTKTest
 3/ 13 Testing SystemInformation
 4/ 13 Testing TestVTKWriters
 5/ 13 Testing TestVTKPython
 6/ 13 Testing VTKPythonMultiGrid
 7/ 13 Testing IronImage
 8/ 13 Testing IronImageMagic
 9/ 13 Testing IronImageStrideMagic
10/ 13 Testing IronRectMagic
11/ 13 Testing IronRectStrideMagic
12/ 13 Testing IronStructMagic
13/ 13 Testing IronStructStrideMagic
```

If we now run

```
ctest -R Python
```

We will only see tests that contain string *Python*:

```
Test project
 1/  3 Testing PythonDataDesc
 2/  3 Testing TestVTKPython
 3/  3 Testing VTKPythonMultiGrid
```

We can also omit tests using -E, for example:

```
ctest -E Iron
```

will produce:

```
Test project
 1/  6 Testing PythonDataDesc
 2/  6 Testing VTKTest
 3/  6 Testing SystemInformation
 4/  6 Testing TestVTKWriters
 5/  6 Testing TestVTKPython
 6/  6 Testing VTKPythonMultiGrid
```

Both -R and -E can be used at the same time.

To determine what tests are avilable, you can always run:

```
ctest -N
```

which will display the list of tests but not actually run them.

The second way of specifying tests is using explicit test number option -I:

```
ctest -I 3,5
```

will run tests:

```
Test project
Running tests: 3 4 5
 3/ 13 Testing SystemInformation
 4/ 13 Testing TestVTKWriters
 5/ 13 Testing TestVTKPython
```

We can also specify stride:

```
ctest -I ,,3
```

will run tests:

```
Test project
Running tests: 1 4 7 10 13
 1/ 13 Testing PythonDataDesc
 4/ 13 Testing TestVTKWriters
 7/ 13 Testing IronImage
10/ 13 Testing IronRectMagic
13/ 13 Testing IronStructStrideMagic
```

Or run individual tests:

```
ctest -I 4,4,,4,7,13
```

will run tests:

```
Test project
Running tests: 4 7 13
 4/ 13 Testing TestVTKWriters
 7/ 13 Testing IronImage
13/ 13 Testing IronStructStrideMagic
```

Make sure that the first and second argument are the index of the first test

Dynamic Analysis

Software development can be significantly hindered when memory leaks are introduced

in the code. Both Purify and Valgrind can catch most of them. Setting up both is extremely easy.

For example, to setup purify, all you have to do is to add:

```
PURIFYCOMMAND:FILEPATH=c:/Progra~1/Rational/common/purify.exe
```

To your cmake cache. Same way to setup valgrind, you add:

```
MEMORYCHECK_COMMAND:FILEPATH=/home/kitware/local/bin/valgrind
```

You can add additional options by specifying MEMORYCHECK_COMMAND_OPTIONS and MEMORYCHECK_SUPPRESSIONS_FILE.

Make sure to run:

```
ctest -D NightlyMemoryCheck
```

or

```
ctest -D NightlyStart
ctest -D NightlyUpdate
ctest -D NightlyConfigure
ctest -D NightlyBuild
ctest -D NightlyTest
ctest -D NightlyMemCheck
ctest -D NightlySubmit
```

Customizing CTest

CTest can be customized by providing CTestCustom.ctest or CTestCustom.cmake file in the *build* tree. If both files exist, CTestCustom.cmake will be preferred. If the CTestCustom.cmake/.ctest file is distributed with the sources of the project, e.g. CONFIGURE_FILE() can be used to put it in the build tree. The file may contain any SET command for any CMake variable, but the following ones will be used:

Variable	Description
CTEST_CUSTOM_ERROR_MATCH	Regular expression for errors during build process
CTEST_CUSTOM_ERROR_EXCEPTION	Regular expression for error exceptions during build process
CTEST_CUSTOM_WARNING_MATCH	Regular expression for warnings during build process

CTEST_CUSTOM_WARNING_EXCEPTION	Regular expression for warning exception during build process
CTEST_CUSTOM_MAXIMUM_NUMBER_OF_ERRORS	Maximum number of errors to display
CTEST_CUSTOM_MAXIMUM_NUMBER_OF_WARNINGS	Maximum number of warnings to display
CTEST_CUSTOM_TESTS_IGNORE	List of tests to ignore during the *Test* stage
CTEST_CUSTOM_MEMCHECK_IGNORE	List of tests to ignore during the *MemCheck* stage
CTEST_CUSTOM_PRE_TEST	Command to execute before any tests are run during *Test* stage
CTEST_CUSTOM_POST_TEST	Command to execute after any tests are run during *Test* stage
CTEST_CUSTOM_MAXIMUM_PASSED_TEST_OUTPUT_SIZE	Maximum size of passed test output
CTEST_CUSTOM_MAXIMUM_FAILED_TEST_OUTPUT_SIZE	Maximum size of failed test output
CTEST_CUSTOM_PRE_MEMCHECK	Command to execute before any tests are run during *MemCheck* stage
CTEST_CUSTOM_POST_MEMCHECK	Command to execute after any tests are run during *MemCheck* stage
CTEST_CUSTOM_COVERAGE_EXCLUDE	Regular expression for excluding files from coverage testing
CTEST_EXTRA_COVERAGE_GLOB	Report on uncovered files matching this expression

Example of CTestCustom.cmake file would be:

```
SET(CTEST_CUSTOM_MEMCHECK_IGNORE
 ${CTEST_CUSTOM_MEMCHECK_IGNORE}
 TestSetGet
 otherPrint-ParaView
 Example-vtkLocal
 Example-vtkMy

 # These tests do not actually run any VTK code
 HeaderTesting-Common
 HeaderTesting-Filtering
 HeaderTesting-Graphics
 HeaderTesting-Imaging
 HeaderTesting-IO

 # this one runs python which then runs two
 # program so no memory checking there
 Sockets-image
 )

SET(CTEST_CUSTOM_WARNING_MATCH
 ${CTEST_CUSTOM_WARNING_MATCH}
```

```
  "{standard input}:[0-9][0-9]*: Warning: "
  )

IF("@CMAKE_SYSTEM@" MATCHES "OSF")
  SET(CTEST_CUSTOM_WARNING_EXCEPTION
    ${CTEST_CUSTOM_WARNING_EXCEPTION}
    "XdmfDOM"
    "XdmfExpr"
    "vtkKWApplication"
    "vtkKWObject"
    )
ENDIF("@CMAKE_SYSTEM@" MATCHES "OSF")

SET(CTEST_CUSTOM_WARNING_EXCEPTION
  ${CTEST_CUSTOM_WARNING_EXCEPTION}
  "tcl8.4.5/[^/]+/../[^/]+/[^.]+.c[:\"]"
  "tk8.4.5/[^/]+/[^/]+.c[:\"]"
  "VTK/Utilities/vtktiff/"
  "Utilities/vtkmpeg2/"
  "Utilities/hdf5/"
  "xtree.[0-9]+. : warning C4702: unreachable code"
  "warning LNK4221"
  "variable .var_args[2]*. is used before its value is set"
  )
```

CTest Scripting

For an example of how CTest can run the whole testing process described in a single script, look at how CMake dashboards are created with the CTest -S script (https://gitlab.kitware.com/cmake/cmake/blob/master/Help/dev/testing.rst).

Conclusion

Performing tests on the project is a great software development practice and can result in significant improvement on the quality of the project. CTest provides a simple and reliable way of performing nightly, continuous, and experimental tests.

More information about CTest can be found in Mastering CMake (http://www.kitware.com/products/cmakebook.html).

- This page was last modified on 30 August 2017, at 11:00.