

CTEST 2.00: an interface for building C++ packages

David L. Adams
Rice University
April 9, 1999

CTEST defines an interface between packages of C++ software and code development environments. Organizing code in the prescribed manner facilitates porting between different experiments and between different platforms within an experiment.

Introduction

Developers of C++ code devote considerable attention to logical design: the choice of classes and their responsibilities, associations and relationships. Typically less attention is paid to the physical design: how classes are assigned to files and files organized into directories, the contents of libraries and how testing is carried out. This physical design is critical for large-scale systems.

Of course, the developer is eventually forced (if only at implementation time) to make the physical design choices. Typically many of these choices are driven by the developer's current desktop environment and conventions of the current experiment. There will likely be a need to reorganize the code when moving to a new environment. This may occur when the current experiment opts to add a new platform or when it is desired to use the code in another experiment or domain.

Here we define a convention for physical organization of both the code and the instructions for building. Developers organize their software in this manner and each development environment is responsible for providing an interface capable following these instructions.

Code organization

We have adopted many of the ideas and much of the terminology in the book by John Lakos: "Large-Scale C++ Software Design" (Addison-Wesley, 1996). Any developer of a large-scale system is encouraged to read this book.

We assume the code is organized into files. The fundamental unit of our software organization is the *component* which consists of a header file, an implementation file and a test file. The second of these contains the source which is compiled into a bare object or library. It includes (through the preprocessor directive `#include`) the header. The test file is

a main program which tests the header interface and implementation source and returns zero if all tests are passed. Typically one class and associated free functions are assigned to a component.

A large-scale system will consist of hundreds or thousands of components and is made more manageable by grouping related components into *packages*. The code and instruction files for each package are contained in a separate directory and each package contributes objects only to its own library. This is also a natural division for management: typically each package is the responsibility of a very small number of developers. A typical package might have five to ten components.

It is often useful to continue the organization by grouping packages into subsystems. Presently CTEST is only defined at the package level. The instructions for building are specific to each package.

In addition to components, a package may include stand-alone headers, additional tests, sources to be compiled into binaries and scripts. The package directory may be divided into subdirectories for purpose of organization or to apply different instructions for different pieces of code. All the library components must reside in the same subdirectory. Everything in the package directory including subdirectories is considered to be part of the package and all files which are part of the package are contained in the directory.

Dependencies

Lakos emphasizes the importance of maintaining acyclic physical dependencies both at the component and package levels. This makes it possible to build in dependency order and test each component and package individually. CTEST does not specify the order in which packages are built but it does specify the order of components within a package.

Packages may depend on one another through included header files or through libraries. Headers in other packages are made available using the standard include mechanism where the first directory in the include file name is the package name.

Build instructions

The instructions for building a package are taken from instruction files within the package. Table 1 lists the actions which are carried out and the instruction files on which each action depends. Those instructions that apply package-wide and appear in top-level package directory are marked with asterisks. The others apply only to the subdirectory in which they reside.

File extensions

There are many conventions for C++ file name extensions and CTEST does not impose one. Instead, the three files HXXTYPE, CXXTYPE and TXXTYPE specify the exten-

Action	Relevant instruction files
Header installation	HEADER_DIR*, HXXTYPE*, INCLUDE_TYPES*, INCLUDE_FILES*, INCLUDES
Library dependencies	LIBDEPS*
Library build	COMPONENTS, CXXTYPE*
Component testing	COMPONENTS, TXXTYPE*, CTEST_DIR*
Object build	OBJECT_COMPONENTS, CXXTYPE*
Integrated testing	ITESTS, OBJECTS, LIBRARIES, CXXTYPE*
Binary build	BINARIES, OBJECTS, LIBRARIES, CXXTYPE*
Script installation	SCRIPTS

Table 1: CTEST actions and associated instruction files. Those marked with asterisks only appear in the top-level package directory.

sions for header, source implementation and test files, respectively. If any of these files is missing, then the respective default `.hpp`, `.cpp` or `_t.cpp` is used. These files may only appear in the top-level package directory and apply to all instructions and all subdirectories.

Processing subdirectories

For all actions except header installation, instructions may reside in the top-level directory or any subdirectory. For the latter actions, each of the subdirectories listed in instruction file `SUBDIRS` is processed after carrying out the action in the current directory.

Header installation

Instructions are provided to publish include files, *i.e.*, specify which files are to be made publicly available for inclusion in other sources. Sources in other packages may include such files as if they were in a subdirectory with the name of the originating package. For example, the C++ header `DemoClass1.hpp` from package `demo` is included as follows:

```
#include "demo/DemoClass1.hpp"
```

An implementation is free to choose how these files are made available: it may make copies, links to individual files or links to directories. It is allowed (but not encouraged) to make unspecified files available in this manner.

All the include files in a directory (and its subdirectories) may be published by listing the directory in `HEADER_DIR`. All files in that package subdirectory with the `HXXTYPE` extension are published. If additional extensions are listed in `INCLUDE_TYPES`, then the corresponding files will also be published. Files in subdirectories of the header directory are made available including that directory path. An implementation might choose to publish the header directory by linking it to the package name in a directory which is part of the compiler include path. An implementation may define a default value to be used if the `HEADER_DIR` instruction is not provided.

More precise control may be obtained by listing individual include files in the top-level instruction `INCLUDE_FILES` or in the local instructions `INCLUDES`. The former lists files including their path relative to the top-level package directory. The latter simply lists files in the local directory. In both cases the file are made available as if they resided directly in the package include area; their actual location in the package hierarchy is not apparent to external packages.

Library dependencies

Each package builds at most one library whose name is the same as that of the package. This library will generally depend on other libraries, *i.e.*, there are other libraries which are used by the objects contained in the package library. These libraries are listed in the top-level instruction `LIBDEPS`. Whenever the package library is used for linking, all of the libraries listed in `LIBDEPS` are listed after it on the link line. This is done recursively so it is not necessary (but is allowed) to list libraries which are dependencies of libraries that are already listed.

Libraries are not allowed to have cyclic dependencies, *i.e.*, if one library depends on another, then the second is not allowed to depend on the first. This applies to both direct (listed in `LIBDEPS`) and indirect (inferred from recursive `LIBDEPS`) dependencies

Library build

The list of library components is taken from instruction files `COMPONENTS`. The extensions specified in `CXXTYPE` are appended to the entries in this list to obtain the respective names of the source files. The sources are compiled and archived in the package library. Depending on the compiler and environment, the list of libraries (derived from `LIBDEPS` for the current package) may also be provided to the archiver.

Components should not acyclic dependencies and should be listed in dependency order, *i.e.*, any component should not depend on those following it in the list. This is typically not relevant for archiving but may be required for component testing.

Components may be distributed among multiple directories with a `COMPONENTS` instruction in each. However there is no guarantee as to the order in which the directories

will be processed so components in different directories should not depend on one another. Typically a package will have only one directory containing components.

A package which has no COMPONENTS files does not build a library.

Component testing

A test file defining a main program must be provided for each component. The names of these files are obtained by appending the value from TXXTYPE to the component name. If the file CTEST_DIR appears in the top-level package directory, then the test files are found in the subdirectory listed in that file. Otherwise (and typically) each test files resides in the same directory as its component source.

The tests are compiled and linked against the package library and its dependent libraries derived from LIBDEPS. The library may only be partially constructed at this point but is guaranteed to contain the current component and all those listed before it in the current COMPONENTS file.

The resulting executable is run using a component test script which is obtained from one of three locations. First the directory containing the component test is checked for a file with the same name as the component and the extension .sh. If that file is not present, then that directory is searched for the file run_component_test.sh. If neither is present, then a default script is used.

Test scripts are described in a later section. This test is considered successful if the script returns zero. The default script runs the test executable and returns its return value.

Object build

The list of object components is taken from OBJECT_COMPONENTS. Again CXXTYPE is appended to each entry to obtain the file names. The files are compiled and the resulting objects are store in a common area available for linking in other packages. Component testing is *not* carried out on the assumption that these components will be simple stubs which force the loading of a library component which has been tested.

Integrated testing

Integrated tests are intended to check behavior which cannot be checked during component testing. This might include interaction with a component or package on which a component is not usually dependent. The list of integrated test sources is taken from instruction file ITESTS. A source file name is obtained by appending CXXTYPE to each entry. The source is compiled and then linked against the objects in OBJECTS and the libraries in LIBRARIES extended using the LIBDEPS mechanism.

As for component tests, this executable is run using a test script which returns zero for success. The script obtained by appending .sh to the test name is used if present or, if not,

then the script `run_integrated_test.sh` is used. If neither is present, then a default script which runs the test and returns its return value is used.

Binary build

The instruction file `BINARIES` contains a list of binaries to build. `CXXTYPE` is appended to each entry to obtain source file name. The source is compiled and the linked against the objects (`OBJECTS`) and libraries (`LIBRARIES` extended with `LIBDEPS`). The resulting executable is installed in a common binary area.

Script installation

Scripts are copied from the package to a common area. The list of such scripts is taken from the instruction file `SCRIPTS`.

Test scripts

Component and integrated tests are run with a test script which is run from a POSIX (sh) shell. The test is considered successful if this script returns zero. A successful test should not write any output to standard out or standard error.

The default scripts run the executable and return its return value. Output written to standard out is redirected to a file. Users may provide their own scripts to provide input files, check output files or carry out any other action.

Test scripts are provided the following arguments: the name of the test executable, the directory containing the sources from which the executable was built, and a list of directories where packages may be found. The first directory in the latter list is the one containing the current package. Figure 1 contains a script fragment: showing how to extract these values.

The directory for the current package `curpkg` is

```
CURDIR=$TOPDIR/curpkg
```

The top-level directory for any package may be obtained with the command `ctpkgpath`. The directory for package `extpkg` is

```
EXTDIR='ctpkgpath extpkg $TOPPATH'
```

It is a requirement on CTEST implementation that they provide the script `ctpkgpath` with this behavior.

```

# First argument is the executable name.
EXE=$1

# Second argument is the source directory.
SRCDIR=$2

# Third argument is the current installation directory.
# If package pkg is installed locally, its top level directory
# is $TOPDIR/pkg.
TOPDIR=$3

# Arguments 3-N specify the search path for packages.
# It consists of directory names separated by spaces.
# The script ctpkgpath will echo the first instance of the
# package directory along the path.
# If not found, it will return nonzero and write to stderr.
shift 2
TOPPATH=$*

```

Figure 1. Test Script fragment showing how to extract command line parameters.

Extending the interface

CTEST includes generic capabilities that we have found generally useful. With the exception of script installation, all apply to the construction of C++ software. We have tried to strike a balance between a very limited interface which would be useful to few developers and an extensive interface which would put a great burden on anyone trying to port the interface to a new environment.

Inevitably, there will be request for additional capabilities. Some of these will be specific to a particular experiment while others may be more generally useful. By its nature, CTEST is easily extended—one only needs to add additional instruction files. Implementations which do not handle the extension will simply ignore these files.

Conclusions

CTEST provides an interface for the organization of C++ code. It is easy to use, supports good physical design principles. Its scope is sufficient for this task but is limited enough to allow implementation in a wide range of code development environments. It is also easily extended if any unanticipated needs should arise.

CTEST versions

Version 2.00

This document describes version 2.00 of the CTEST interface. The major changes are the addition of the LIBDEPS mechanism and the extension of the specification of include files. In addition, the structure of test scripts has been clarified. Except for the title and this paragraph, it is the same as the April 9 draft document.

Version 1

The original interface was presented at CHEP98.

Further information

For updates and further information about CTEST, please see the CTEST home page at <http://www.bonner.rice.edu/adams/ctest>.