CTEST: an interface for building C++ packages

David L. Adams
Rice University
October 12, 1998

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 though 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. In general, instructions only apply to the subdirectory in which they reside. An exception to this are the file extension instructions described in the following section.

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
Library build (and component testing)	COMPONENTS, LIBRARIES
Object build	OBJECT_COMPONENTS
Integrated testing	ITESTS, OBJECTS, LIBRARIES
Binary build	BINARIES, OBJECTS, LIBRARIES
Script installation	SCRIPTS

Table 1: CTEST actions and associated instruction files.

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

The location of public header files is taken from the instruction file HEADER_DIR in the top-level directory. Files with extensions specified in HXXTYPE are made available as though they resided in a local directory with the name of the package. This applies anywhere in the subdirectory tree. This may be accomplished by copying the headers to an appropriate area or defining the header path to point into the package areas.

Library build

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

The tests are compiled and linked against this library (possibly only partially constructed) and the libraries listed in LIBRARIES. The entries in LIBRARIES are package names. The test is run and the builder is notified of the result.

The components used to build the library should all appear in one subdirectory. *I.e.*, the file COMPONENTS only appears in one place.

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, linked against the objects in OBJECTS and the libraries in LIBRARIES and run. The test returns zero for success and the builder is notified of the result.

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). 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 actually run with a test script which echoes the return value of the test. Developers may substitute their own file for this test script. This allows one to copy over files before testing, provide command line input to the test or verify the existence or contents of output files. The script should return zero to indicate success.

CTEST first searches the current directory for a component- or test-specific script and then for a package-specific (actually subdirectory-specific) version before using the default. The component- or test-specific script name is obtained by appending .sh to the component or test name. The package-specific scripts are called run_component_test.sh and run_integrated_test.sh.

These and any user-supplied scripts are written in the POSIX scripting language (sh).

Within a test script, it may be desirable to obtain the location of another package. This may be accomplished with the command ctpkgpath.

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.

Further information

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