Generic Build Instructions

Setup

To build Google Test and your tests that use it, you need to tell your build system where to find its headers and source files. The exact way to do it depends on which build system you use, and is usually straightforward.

Build

Suppose you put Google Test in directory `${GTEST_DIR}`. To build it, create a library build target (or a project as called by Visual Studio and Xcode) to compile

```
${GTEST_DIR}/src/gtest-all.cc
```

with `${GTEST_DIR}/include` in the system header search path and `${GTEST_DIR}` in the normal header search path. Assuming a Linux-like system and gcc, something like the following will do:

```
g++ -isystem ${GTEST_DIR}/include -I${GTEST_DIR} \
    -pthread -c ${GTEST_DIR}/src/gtest-all.cc
ar -rv libgtest.a gtest-all.o
```

(We need `-pthread` as Google Test uses threads.)

Next, you should compile your test source file with `${GTEST_DIR}/include` in the system header search path, and link it with gtest and any other necessary libraries:

```
g++ -isystem ${GTEST_DIR}/include -pthread path/to/your_test.cc libgtest.a \
    -o your_test
```

As an example, the make/ directory contains a Makefile that you can use to build Google Test on systems where GNU make is available (e.g. Linux, Mac OS X, and Cygwin). It doesn't try to build Google Test's own tests. Instead, it just builds the Google Test library and a sample test. You can use it as a starting point for your own build script.

If the default settings are correct for your environment, the following commands should succeed:

```
cd ${GTEST_DIR}/make
make
./sample1_unittest
```

If you see errors, try to tweak the contents of `make/Makefile` to make them go away. There are instructions in `make/Makefile` on how to do it.

Using CMake

Google Test comes with a CMake build script (CMakeLists.txt) that can be used on a wide range of platforms ("C" stands for cross-platform.). If you don't have CMake installed already, you can download it for free from http://www.cmake.org/.

CMake works by generating native makefiles or build projects that can be used in the compiler environment of your choice. The typical workflow starts with:

```
mkdir mybuild       # Create a directory to hold the build output.
cd mybuild
cmake ${GTEST_DIR}  # Generate native build scripts.
```

If you want to build Google Test's samples, you should replace the last command with

```
cmake -Dgtest_build_samples=ON ${GTEST_DIR}
```

If you are on a *nix system, you should now see a Makefile in the current directory. Just type 'make' to build gtest.

If you use Windows and have Visual Studio installed, a `gtest.sln` file and several `.vcproj` files will be created. You can then build them using Visual Studio.

On Mac OS X with Xcode installed, a `.xcodeproj` file will be generated.

Legacy Build Scripts

Before settling on CMake, we have been providing hand-maintained build projects/scripts for Visual Studio, Xcode, and Autotools. While we continue to provide them for convenience, they are not actively maintained any more. We highly recommend that you follow the instructions in the previous two sections to integrate Google Test with your existing build system.

If you still need to use the legacy build scripts, here's how:

The msvc\ folder contains two solutions with Visual C++ projects. Open the `gtest.sln` or `gtest-md.sln` file using Visual Studio, and you are ready to build Google Test the same way you build any Visual Studio project. Files that have names ending with -md use DLL versions of Microsoft runtime libraries (the /MD or the /MDd compiler option). Files without that suffix use static versions of the runtime libraries (the /MT or the /MTd option). Please note that one must use the same option to compile both gtest and the test code. If you use Visual Studio 2005 or above, we recommend the -md version as /MD is the default for new projects in these versions of Visual Studio.

On Mac OS X, open the `gtest.xcodeproj` in the `xcode/` folder using Xcode. Build the "gtest" target. The universal binary framework will end up in your selected build directory (selected in the Xcode "Preferences..." -> "Building" pane and defaults to xcode/build). Alternatively, at the command line, enter:

```
xcodebuild
```

This will build the "Release" configuration of gtest.framework in your default build location. See the "xcodebuild" man page for more information about building different configurations and building in different locations.

If you wish to use the Google Test Xcode project with Xcode 4.x and above, you need to either:

- update the SDK configuration options in xcode/Config/General.xconfig. Comment options `SDKROOT`, `MACOS_DEPLOYMENT_TARGET`, and `GCC_VERSION`. If you choose this route you lose the ability to target earlier versions of MacOS X.
- Install an SDK for an earlier version. This doesn't appear to be supported by Apple, but has been reported to work (http://stackoverflow.com/questions/5378518).

Tweaking Google Test

Google Test can be used in diverse environments. The default configuration may not work (or may not work well) out of the box in some environments. However, you can easily tweak Google Test by defining control macros on the compiler command line. Generally, these macros are named like `GTEST_XYZ` and you define them to either 1 or 0 to enable or disable a certain feature.

We list the most frequently used macros below. For a complete list, see file include/gtest/internal/gtest-port.h.

Choosing a TR1 Tuple Library

Some Google Test features require the C++ Technical Report 1 (TR1) tuple library, which is not yet available with all compilers. The good news is that Google Test implements a subset of TR1 tuple that's enough for its own need, and will automatically use this when the compiler doesn't provide TR1 tuple.

Usually you don't need to care about which tuple library Google Test uses. However, if your project already uses TR1 tuple, you need to tell Google Test to use the same TR1 tuple library the rest of your project uses, or the two tuple implementations will clash. To do that, add

```
-DGTEST_USE_OWN_TR1_TUPLE=0
```

to the compiler flags while compiling Google Test and your tests. If you want to force Google Test to use its own tuple library, just add

```
-DGTEST_USE_OWN_TR1_TUPLE=1
```

to the compiler flags instead.

If you don't want Google Test to use tuple at all, add

```
-DGTEST_HAS_TR1_TUPLE=0
```

and all features using tuple will be disabled.

Multi-threaded Tests

Google Test is thread-safe where the pthread library is available. After `#include "gtest/gtest.h"`, you can check the `GTEST_IS_THREADSAFE` macro to see whether this is the case (yes if the macro is `#defined` to 1, no if it's undefined.).

If Google Test doesn't correctly detect whether pthread is available in your environment, you can force it with

```
-DGTEST_HAS_PTHREAD=1
```

or

```
-DGTEST_HAS_PTHREAD=0
```

When Google Test uses pthread, you may need to add flags to your compiler and/or linker to select the pthread library, or you'll get link errors. If you use the CMake script or the deprecated Autotools script, this is taken care of for you. If you use your own build script, you'll need to read your compiler and linker's manual to figure out what flags to add.

As a Shared Library (DLL)

Google Test is compact, so most users can build and link it as a static library for the simplicity. You can choose to use Google Test as a shared library (known as a DLL on Windows) if you prefer.

To compile *gtest* as a shared library, add

```
-DGTEST_CREATE_SHARED_LIBRARY=1
```

to the compiler flags. You'll also need to tell the linker to produce a shared library instead - consult your linker's manual for how to do it.

To compile your *tests* that use the gtest shared library, add

```
-DGTEST_LINKED_AS_SHARED_LIBRARY=1
```

to the compiler flags.

Note: while the above steps aren't technically necessary today when using some compilers (e.g. GCC), they may become necessary in the future, if we decide to improve the speed of loading the library (see http://gcc.gnu.org/wiki/Visibility for details). Therefore you are recommended to always add the above flags when using Google Test as a shared library. Otherwise a future release of Google Test may break your build script.

Avoiding Macro Name Clashes

In C++, macros don't obey namespaces. Therefore two libraries that both define a macro of the same name will clash if you `#include` both definitions. In case a Google Test macro clashes with another library, you can force Google Test to rename its macro to avoid the conflict.

Specifically, if both Google Test and some other code define macro FOO, you can add

```
-DGTEST_DONT_DEFINE_FOO=1
```

to the compiler flags to tell Google Test to change the macro's name from `FOO` to `GTEST_FOO`. Currently `FOO` can be `FAIL`, `SUCCEED`, or `TEST`. For example, with `-DGTEST_DONT_DEFINE_TEST=1`, you'll need to write

```
GTEST_TEST(SomeTest, DoesThis) { ... }
```

instead of

```
TEST(SomeTest, DoesThis) { ... }
```

in order to define a test.

Developing Google Test

This section discusses how to make your own changes to Google Test.

Testing Google Test Itself

To make sure your changes work as intended and don't break existing functionality, you'll want to compile and run Google Test's own tests. For that you can use CMake:

```
mkdir mybuild
cd mybuild
cmake -Dgtest_build_tests=ON ${GTEST_DIR}
```

Make sure you have Python installed, as some of Google Test's tests are written in Python. If the cmake command complains about not being able to find Python (`Could NOT find PythonInterp (missing: PYTHON_EXECUTABLE)`), try telling it explicitly where your Python executable can be found:

```
cmake -DPYTHON_EXECUTABLE=path/to/python -Dgtest_build_tests=ON ${GTEST_DIR}
```

Next, you can build Google Test and all of its own tests. On *nix, this is usually done by 'make'. To run the tests, do

```
make test
```

All tests should pass.

Normally you don't need to worry about regenerating the source files, unless you need to modify them. In that case, you should modify the corresponding .pump files instead and run the pump.py Python script to regenerate them. You can find pump.py in the scripts/ directory. Read the Pump manual for how to use it.