

```
1
2 example: example.cpp example_main.cpp
3     g++ -Wall -std=c++11 example.cpp example_main.cpp -o example
4
```

```
class example_test : public ::testing::Test {
protected:
    example_test() { }
    virtual ~example_test() { }
    virtual void SetUp() { }
    virtual void TearDown() { }
};

TEST_F(example_test, test_basic) {
    ASSERT_EQ(13, 10 + 3);
}
```

Makefiles and Testing

Matthew Mussomele

What are makefiles?

Makefiles are a special kind of file that essentially tell the computer how to compile your code.

They allow you to organize and automate your compilation commands.

In addition, makefiles will only run the compile commands they actually need to.

If you haven't modified a file since it's last compilation, it won't be compiled again.

Makefile Structure

Makefile entries have the following structure:

```
<target>: <dependency> <dependency> ...  
    <command>
```

Target - The 'name' of the entry

Dependency - The targets or files that the target depends on

Command - The command to be run when this target
 needs updating

Makefile Example

Suppose that we have the following project structure:

- ❖ List/

- ❖ list.cpp

- ❖ list.h

- ❖ Sorting/

- ❖ sorts.cpp

- ❖ sorts.h

The List and Sorting folders both represent projects we are working on.

The List folder contains an implementation of some kind of list structure.

The Sorting folder contains an implementation of some list sorting algorithms.

Makefile Example

A makefile for that project could look like this:

```
2  # target 'all' is dependent on the targets sorting.o and list.o
3  # if either requires recompilation, 'all' also recompiles after
4  # building it's dependencies
5  all: sorting.o list.o
6      g++ sorting.o list.o -o sorting_example
7
8  # target 'sorting.o' is dependent on the Sorting folder's source code
9  # if either file has changed since the last build or no build exists,
10 # the g++ command will be ran
11 sorting.o: Sorting/sorts.cpp Sorting/sorts.h List/list.h
12     g++ -c sorts.cpp
13
14 # this is the same as the 'sorting.o' target except for the List folder
15 list.o: List/list.cpp List/list.h
16     g++ -c list.cpp
```

Why Use Makefiles?

Makefiles are very useful when building a project that has multiple files.

It allows you to simply run the makefiles instead of repetitively running several `g++` commands every time you want to compile.

It will also only build the files that it has to, saving time and resources on larger projects.

Testing

Unit testing is very important to do when developing software.

It allows you to:

- ❖ Set a standard for behavior of your software
- ❖ Make changes and quickly verify that you didn't break anything
- ❖ Narrow down bugs to specific parts of your program.

Important Things When Testing

Make sure to test the smallest parts of your program that you can, like testing every function that seems breakable.

A function that simply returns a number or adds two things probably isn't breakable.

Try to cover as many different situations as you can in your tests.

Start with expected, common inputs and then test edge cases and unlikely inputs as well.

Google Test

This summer, we are using the Google Test C++ testing framework.

It allows for comprehensive and clear testing of functions, classes and more.

It also gives you the ability to verify that your code is correct so you don't have to manually check that it works.

Making Google Test Files

Simple Google Test files have 3 main components.

1. A test class declaration (just copy and paste this, you don't need to understand it)
2. A suite of tests, each testing a different behavior of your program
3. A main function that runs all the tests (you can just copy-paste this too)

It's good to have several tests for each function, with different tests covering different cases so you can quickly know what went wrong.

The Test Class

The Google Test test class is how Google Test knows to run your tests.

For simplicity's sake, we'll just give you a very basic test class; there's no need to do anything with it.

```
namespace { // don't worry about this

    class example_test : public ::testing::Test {
    protected:
        example_test() { }
        virtual ~example_test() { }
        virtual void SetUp() { }
        virtual void TearDown() { }
    };

    /* YOUR TESTS HERE */

}
```


The Tests

Tests are simple structures.

They have a header, parameters and a body, just like functions.

Tests do not have return types.

Use the TEST_F macro as the 'name' of every test.

```
namespace { // don't worry about this

class example_test : public ::testing::Test {
protected:
    example_test() { }
    virtual ~example_test() { }
    virtual void SetUp() { }
    virtual void TearDown() { }
};

/* YOUR TESTS HERE */
TEST_F(example_test, test_addition) {
    ASSERT_EQ(13, 10 + 3);
}

}
```

Test class name, must match

Test name, can be any valid name

Main

A Google Test main function is a very simple structure, but you aren't expected to understand exactly what's happening.

Simply copy and paste it at the bottom, outside the namespace block of a test file.

```
int main(int argc, char ** argv) {  
    ::testing::InitGoogleTest(&argc, argv);  
    return RUN_ALL_TESTS();  
}
```


A Complete Test File

```
#include "gtest/gtest.h"

namespace { // don't worry about this

    class example_test : public ::testing::Test {
    protected:
        example_test() { }
        virtual ~example_test() { }
        virtual void SetUp() { }
        virtual void TearDown() { }
    };

    /* YOUR TESTS HERE */
    TEST_F(example_test, test_addition) {
        ASSERT_EQ(13, 10 + 3);
    }

}

int main(int argc, char ** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Assertions & Expectations

Assertions and expectations are the foundation of testing.

Assertions allow you to check if some calculated value is equal to what you expected it to be.

A failed assertion will cause a test to stop execution.

Expectations allow the same thing, but failing an expectation will not stop the test execution.

Most assertions and expectations have the structure:

```
ASSERT_EQ(<expected>, <actual>)
```

A List of Assertions

- ❖ ASSERT_TRUE
- ❖ ASSERT_EQ (equal)
- ❖ ASSERT_LT (less than)
- ❖ ASSERT_GT (greater than)
- ❖ ASSERT_STREQ (string eq)
- ❖ ASSERT_STRCASEEQ (string equals, ignore case)
- ❖ ASSERT_FALSE
- ❖ ASSERT_NE (not equal)
- ❖ ASSERT_LE (less than or equal to)
- ❖ ASSERT_GE (greater than or equal to)
- ❖ ASSERT_STRNE (string not equals)
- ❖ ASSERT_STRCASENE (string not equals, ignore case)

Expectations are the same, just replace ASSERT with EXPECT

Building Google Test

You may have noticed that the main function for your tasks are always in a separate file from the functions they use.

This is because if it wasn't it would conflict with the Google Test main whenever you wanted to compile tests.

Instead we keep them separate so two versions can be built:

1. A regular build, using your task file and the main file
2. A test build, using your task file and the test file

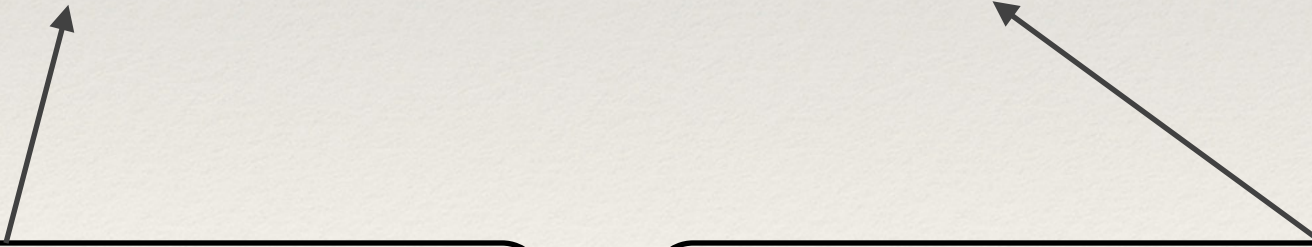
Building Google Test

Building a test file isn't as easy as regular compiling.

You need to add linking commands so the compiler knows where to find the Google Test files that handle the actual running of the tests.

Use the following command:

```
g++ -std=c++11 <your files> -lgtest -o <output_filename>
```



Use the 2011 gcc compiler features
(you should do this anyway)

Link the Google Test libraries when compiling

Running Google Test

You can run a test file like any other compiled C++ file.

It will execute, running all the tests that it finds and printing out useful information about their success / failure.

Google Test Output

A 100% score on the Functions task

A Functions attempt with a slightly incorrect `absolute_sum` function

```
[=====] Running 10 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 10 tests from function_test
[ RUN    ] function_test.test_print_joined_basic
[ OK     ] function_test.test_print_joined_basic (0 ms)
[ RUN    ] function_test.test_print_joined_neg_basic
[ OK     ] function_test.test_print_joined_neg_basic (0 ms)
[ RUN    ] function_test.test_print_joined_no_middle_neg
[ OK     ] function_test.test_print_joined_no_middle_neg (0 ms)
[ RUN    ] function_test.test_print_joined_no_overflow
[ OK     ] function_test.test_print_joined_no_overflow (0 ms)
[ RUN    ] function_test.test_absolute_sum_basic
[ OK     ] function_test.test_absolute_sum_basic (0 ms)
[ RUN    ] function_test.test_absolute_sum_correct_abs
[ OK     ] function_test.test_absolute_sum_correct_abs (0 ms)
[ RUN    ] function_test.test_largest_product_irrelevant_order
[ OK     ] function_test.test_largest_product_irrelevant_order (0 ms)
[ RUN    ] function_test.test_largest_product_duplicates
[ OK     ] function_test.test_largest_product_duplicates (0 ms)
[ RUN    ] function_test.test_fact_or_fib_fact
[ OK     ] function_test.test_fact_or_fib_fact (0 ms)
[ RUN    ] function_test.test_fact_or_fib_fib
[ OK     ] function_test.test_fact_or_fib_fib (0 ms)
[-----] 10 tests from function_test (0 ms total)

[-----] Global test environment tear-down
[=====] 10 tests from 1 test case ran. (0 ms total)
[ PASSED ] 10 tests.
```

```
[=====] Running 10 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 10 tests from function_test
[ RUN    ] function_test.test_print_joined_basic
[ OK     ] function_test.test_print_joined_basic (0 ms)
[ RUN    ] function_test.test_print_joined_neg_basic
[ OK     ] function_test.test_print_joined_neg_basic (0 ms)
[ RUN    ] function_test.test_print_joined_no_middle_neg
[ OK     ] function_test.test_print_joined_no_middle_neg (0 ms)
[ RUN    ] function_test.test_print_joined_no_overflow
[ OK     ] function_test.test_print_joined_no_overflow (0 ms)
[ RUN    ] function_test.test_absolute_sum_basic
[ OK     ] function_test.test_absolute_sum_basic (0 ms)
[ RUN    ] function_test.test_absolute_sum_correct_abs
functions_test.cpp:61: Failure
Value of: absolute_sum(-5, 15)
Actual: 20
Expected: 10
[ FAILED ] function_test.test_absolute_sum_correct_abs (0 ms)
[ RUN    ] function_test.test_largest_product_irrelevant_order
[ OK     ] function_test.test_largest_product_irrelevant_order (0 ms)
[ RUN    ] function_test.test_largest_product_duplicates
[ OK     ] function_test.test_largest_product_duplicates (0 ms)
[ RUN    ] function_test.test_fact_or_fib_fact
[ OK     ] function_test.test_fact_or_fib_fact (0 ms)
[ RUN    ] function_test.test_fact_or_fib_fib
[ OK     ] function_test.test_fact_or_fib_fib (0 ms)
[-----] 10 tests from function_test (0 ms total)

[-----] Global test environment tear-down
[=====] 10 tests from 1 test case ran. (0 ms total)
[ PASSED ] 9 tests.
[ FAILED ] 1 test, listed below:
[ FAILED ] function_test.test_absolute_sum_correct_abs

1 FAILED TEST
```

Citations

- ❖ "Googletest - Google C++ Testing Framework - Google Project Hosting." Googletest - Google C++ Testing Framework - Google Project Hosting. Google, Web. 11 June 2015.