EECS 211 Make Notes

Projects and makefiles

Real programs, even the simple ones we do in EECS 211, are built by compiling multiple source files and linking in multiple library files. While you can do this in one long call to `g++`, it's pretty tedious and error-prone to have to type out such things. Furthermore, most of the time, when you have several source files, only a few need to be recompiled, so compiling all of them is a waste of time.

This is why `make` was invented. `make` is a program that lets you specify in a text file, usually called `Makefile` (note the capitalization), exactly how to build an application.

> *In an integrated development environment (IDE) for C++ or Java or C#, the corresponding concept is usually called a project. You specify in the IDE what files are in the project, what libraries and compiler switches to use, and so on. Internally, the IDE creates either a makefile, or something equivalent to a makefile. The **build** command in an IDE is the equivalent of calling `make` in Unix. A similar modern utility is Ant. Ant is very often used in Java application development.*

The syntax of Makefiles developed over time to handle many different tasks involved in compiling and deploying large applications. Common repeated activities were given shorthand notations. These can make Makefiles very short but also very cryptic. The Makefile template presented here should be sufficient for this course.

Making a project with make

Create a directory for all of your EECS 211 projects. Avoid directory name with spaces, like `My Documents`.

Download and install CppUnit. You'll need it for the example project below and the exercises. This example project willl help test your CppUnit setup.

Download example.zip. Extract the folder it contains into your 211 directory. This should give you one directory called `example` with two files, `Makefile` and `ExampleTests.cpp`.

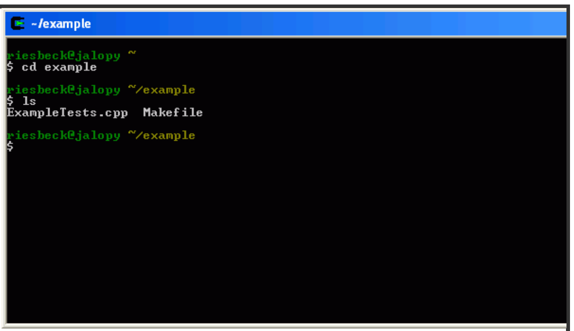
Open a Unix terminal window:

- On Windows, open a Cygwin terminal window. Do not use a normal Windows command shell.
- On MacOS X, start the Terminal application.

Cygwin screenshots will be used here, but the contents will be the same on any Unix-like platform.

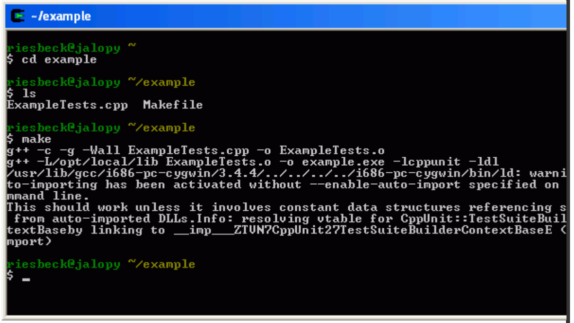
Click on any screenshot to see a larger version.

`cd` to the `example` directory. Do `ls` to make sure you see the two source files.



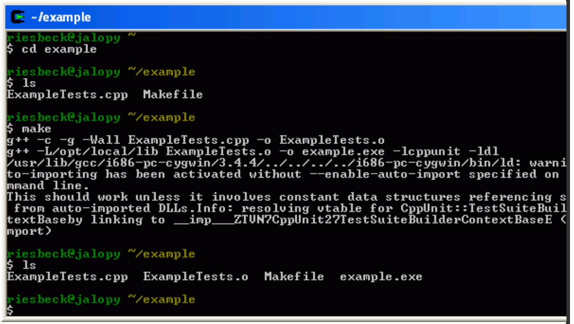
```
~/example
riesbeck@jalopy ~
$ cd example

riesbeck@jalopy ~/example
$ ls
ExampleTests.cpp  Makefile

riesbeck@jalopy ~/example
$
```

Do `make` to run the Makefile. You should see a few messages as the code is compiled and linked. (In Cygwin, there will be a warning about `enable-auto-import` that you can ignore.) If you see a lot of error messages, something is not installed correctly.



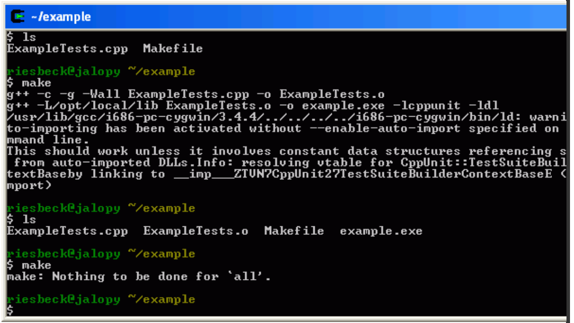
```
~/example
riesbeck@jalopy ~
$ cd example

riesbeck@jalopy ~/example
$ ls
ExampleTests.cpp  Makefile

riesbeck@jalopy ~/example
$ make
g++ -c -g -Wall ExampleTests.cpp -o ExampleTests.o
g++ -L/opt/local/lib ExampleTests.o -o example.exe -lcppunit -ldl
/usr/lib/gcc/i686-pc-cygwin/3.4.4/../../../../i686-pc-cygwin/bin/ld: warni
to-importing has been activated without --enable-auto-import specified on
mmand line.
This should work unless it involves constant data structures referencing s
 from auto-imported DLLs.Info: resolving vtable for CppUnit::TestSuiteBuil
textBaseby linking to __imp___ZTVN7CppUnit27TestSuiteBuilderContextBaseE (
mport)

riesbeck@jalopy ~/example
$ _
```

Do `ls`. You should see an object file for `ExampleTests` (the compiled version of that file), plus the final application file, `example.exe`. (For uniformity over all three platforms, the Makefile creates applications with the `.exe` extension, even though only Windows needs it.)



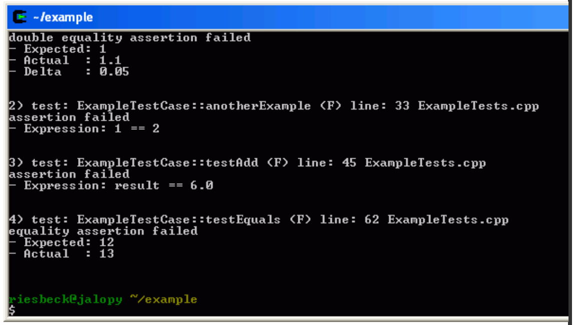
```
~/example
riesbeck@jalopy ~
$ cd example

riesbeck@jalopy ~/example
$ ls
ExampleTests.cpp  Makefile

riesbeck@jalopy ~/example
$ make
g++ -c -g -Wall ExampleTests.cpp -o ExampleTests.o
g++ -L/opt/local/lib ExampleTests.o -o example.exe -lcppunit -ldl
/usr/lib/gcc/i686-pc-cygwin/3.4.4/../../../../i686-pc-cygwin/bin/ld: warni
to-importing has been activated without --enable-auto-import specified on
mmand line.
This should work unless it involves constant data structures referencing s
 from auto-imported DLLs.Info: resolving vtable for CppUnit::TestSuiteBuil
textBaseby linking to __imp___ZTVN7CppUnit27TestSuiteBuilderContextBaseE (
mport)

riesbeck@jalopy ~/example
$ ls
ExampleTests.cpp  ExampleTests.o  Makefile  example.exe

riesbeck@jalopy ~/example
$
```

Do `make` again. You should see a message saying that nothing needs to be done. The rules in the Makefile only run if relevant files have changed since the last time the application was built.



```
~/example
$ ls
ExampleTests.cpp  Makefile

riesbeck@jalopy ~/example
$ make
g++ -c -g -Wall ExampleTests.cpp -o ExampleTests.o
g++ -L/opt/local/lib ExampleTests.o -o example.exe -lcppunit -ldl
/usr/lib/gcc/i686-pc-cygwin/3.4.4/../../../../i686-pc-cygwin/bin/ld: warni
to-importing has been activated without --enable-auto-import specified on
mmand line.
This should work unless it involves constant data structures referencing s
 from auto-imported DLLs.Info: resolving vtable for CppUnit::TestSuiteBuil
textBaseby linking to __imp___ZTVN7CppUnit27TestSuiteBuilderContextBaseE (
mport)

riesbeck@jalopy ~/example
$ ls
ExampleTests.cpp  ExampleTests.o  Makefile  example.exe

riesbeck@jalopy ~/example
$ make
make: Nothing to be done for `all'.

riesbeck@jalopy ~/example
$
```

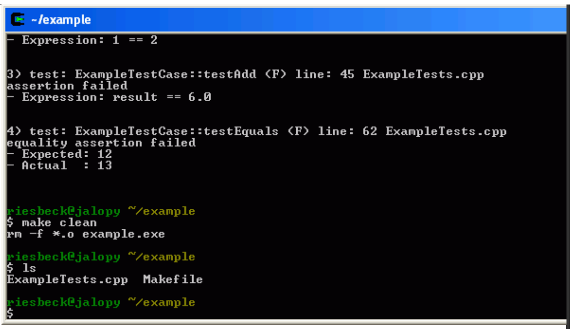
Run the application by typing `./example.exe`. You should see messages about some tests failing. This is correct behavior. `ExampleTests.cpp` has tests that are supposed to fail, so that you can see what test failures look like.



```
~/example
double equality assertion failed
- Expected: 1
- Actual  : 1.1
- Delta   : 0.05


2) test: ExampleTestCase::anotherExample (F) line: 33 ExampleTests.cpp
assertion failed
- Expression: 1 == 2


3) test: ExampleTestCase::testAdd (F) line: 45 ExampleTests.cpp
assertion failed
- Expression: result == 6.0


4) test: ExampleTestCase::testEquals (F) line: 62 ExampleTests.cpp
equality assertion failed
- Expected: 12
- Actual  : 13



riesbeck@jalopy ~/example
$
```

Finally, type `make clean`. This runs `make` with the target `clean`, instead of the default target `all`. You should see a command that removes all the files created by compilation.

Verify that the new files have been removed by doing another `ls`.



```
- Expression: 1 == 2

3) test: ExampleTestCase::testAdd (F) line: 45 ExampleTests.cpp
assertion failed
- Expression: result == 6.0

4) test: ExampleTestCase::testEquals (F) line: 62 ExampleTests.cpp
equality assertion failed
- Expected: 12
- Actual  : 13


riesbeck@jalopy ~/example
$ make clean
rm -f *.o example.exe

riesbeck@jalopy ~/example
$ ls
ExampleTests.cpp  Makefile

riesbeck@jalopy ~/example
$
```

Making Makefiles for new EECS 211 Projects

To adapt the example Makefile for other EECS 211 projects, you only need to change the first few lines of the example Makefile. Here's the Makefile, with the text to change *shown in this style*.

> Use the example Makefile in the Zip archive. Do not copy and paste text from this web page. The indented lines in a Makefile must be indented with tab characters, which can easily be lost by copying from a web page. See the warnings about how to copy and edit this file.

```
Edit this line to describe your project
# ExampleTests Project
Edit this line to list your C++ files (not header files)
SRCS = ExampleTests.cpp
Edit this line to list your header files -- there were none in the example
HDRS =
Edit this line to name your application
PROJ = example

# Remaining lines shouldn't need changing
# Here's what they do:
#   - rebuild if any header file changes
#   - include CppUnit as dynamic library
#   - search /opt/local for MacPorts
#   - generate .exe files for Windows
#   - add -enable-auto-import flag for Cygwin only

First, define variables for the make rules. Many of these are common names.
CC = g++       Use g++ to compile
OBJS = $(SRCS:.cpp=.o)       Make a list of .o files from the list of .cpp files
APP = $(PROJ).exe       Make the name of the executable
CFLAGS = -c -g -Wall       Compiler flags
ifeq (,$(findstring CYGWIN,$(shell uname))) Test for Cygwin
  LDFLAGS = -L/opt/local/lib       Linker flags
else
  LDFLAGS = -L/opt/local/lib -enable-auto-import Cygwin needs this flag
endif
LIBS = -lcppunit -ldl       Libraries to link into the application

all: $(APP)       "make all" is the default target

$(APP): $(OBJS)       make the executable if any object file needed updating
        $(CC) $(LDFLAGS) $(OBJS) -o $(APP) $(LIBS)       Assemble the executable.

%.o: %.cpp $(HDRS)       make file.o if file.cpp or any header changed
        $(CC) $(CFLAGS) $< -o $@       compiles file.cpp to file.o
```

```
clean:      the rule for "make clean"
        rm -f *.o $(APP)      remove object files and the executable
```

To learn more about how `make` works and what you can do with it, see the GNU make manual.

Tips

When debugging Makefile's, the following two flags can be handy:

- `-n` -- this causes `make` to only print the actions it would do, but not actually do them
- `-B` -- this forces `make` to run all the actions that apply, whether or not files have changed

Therefore, an easy way to "dry run" a Makefile is to type:

```
make -B -n
```

This will show the commands that the Makefile will do to build a project from scratch.

WARNINGS

Make sure that all indented command lines **start with a tab character**. This is the single most common problem with makefiles. If the indented lines start with spaces, `make` will mis-interpret them.

In any other file, e.g., source code, avoid tab characters. Turn them off in your editor. There is no standard amount for indenting a line with a tab character. It depends on the setting of the editor. Some emailers erase tab characters. Nicely laid out code with tab characters will most likely be an unreadable mess when looked at by someone else.