Makefile Tutorial

This file is derived from a tutorial originally created by Hector Urtubia.

Compiling your source code files can be tedious, especially when you want to include several source files and have to type the compiling command every time you want to do it. Makefiles are special format files that together with the *make* utility will help you to automatically build and manage your projects.

The `make` utility

To invoke make, simply type:

```
make
```

This program will look for a file named `Makefile` in your directory and execute it.

If you have several makefiles, then you can execute them with the command:

```
make -f MyMakefile
```

There are several other switches to the `make` utility. For more info, `man make`.

Makefile rules

A makefile primarily consisting of rules formatted like this:

```
target: dependencies
[tab] system command
```

The basic Makefile

The trivial way to compile the files and obtain an executable, is by running the command:

```
g++ main.cpp hello.cpp factorial.cpp -o hello
```

To automatically execute this command, you can create a simple Makefile with these contents:

```
all:
        g++ main.cpp hello.cpp factorial.cpp -o hello
```

On this first example we see that our target is called *all*. This is the default target for makefiles. The *make* utility will execute this target if no other one is specified. We also see that there are no dependencies for target *all*, so *make* will execute the specified system command(s). In this case, the command compiles the program according to the command line we gave it.

Using Dependencies

For larger projects, it can be helpful to use different targets. This is because if you modify a single file in your project, you don't have to recompile everything, only what you modified. To accomplish this, it requires breaking the build process into two steps: a. Compilation of source code files into an object file. b. Linking the object files into an executable.

Here is an example:

```
all: hello

hello: main.o factorial.o hello.o
	g++ main.o factorial.o hello.o -o hello

main.o: main.cpp
	g++ -c main.cpp

factorial.o: factorial.cpp
	g++ -c factorial.cpp

hello.o: hello.cpp
	g++ -c hello.cpp

clean:
	rm -rf *.o hello
```

Now we see that the target *all* has only dependencies, but no system commands. In order for *make* to execute correctly, it has to meet all the dependencies of the called target (in this case *all*). Each of the dependencies are searched through all the targets available and executed if found. In this example we see a target called *clean*. It is useful to have such target if you want to have a fast way to get rid of all the object files and executables.

Using Variables

You can also use variables when writing Makefiles. It comes in handy in situations where you want to change the compiler, or the compiler options.

```
CC=g++
CFLAGS=-c -Wall

all: hello

hello: main.o factorial.o hello.o
	$(CC) main.o factorial.o hello.o -o hello

main.o: main.cpp
	$(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
	$(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
	$(CC) $(CFLAGS) hello.cpp
```

```
clean:
	rm -rf *.o hello
```

As you can see, variables can be very useful sometimes. To use them, just assign a value to a variable before you start to write your targets. After that, you can just use them with the dereference operator $(VAR).

More Advanced Makefiles

Here is a more sophisticated Makefile that compiles all of the source code files with the same flags. This Makefile can be reused for other C++ programs by merely modifying the source code files and executable name. Fully understanding this example, requires knowledge of variable substitutions, special variables (such as `$@`) and special targets (such as `.cpp.o`).

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp factorial.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:
	$(CC) $(CFLAGS) $< -o $@

clean:
	rm -rf $(OBJECTS) $(EXECUTABLE)
```

More Information

There is much more to the make facility than what it is listed here. For more information, consult the GNU documentation. (http://www.gnu.org/software/make/manual/make.html)

One known deficiency with this tutorial is that it does not address header files (.h). The GNU documentation has examples on how to handle header files.