Practicum 3 – Make

Assignment Details

Assigned: February 4th, 2013. Due: February 6th, 2013 at midnight.

Background

This assignment will provide an introduction to the `make` utility and instructions for creating a simple `makefile` – note that a simple `makefile` will be required for PA1 part II.

So what exactly is make?

`make` is a standard tool in Linux which automates part of the build process. The total number of things it can do is actually pretty huge, but we're just going to cover a couple of things here.

It should be noted that `make` itself is fairly ancient – dates back to 1977 – and as a result has some strange ideas about syntax and such, but it's still a very powerful tool (powerful enough that even though it has a lot of odd conventions, it's still around).

The Practicum

Ok, to get started, log in to your multilab account as normal.

Instead of using a specific folder for the practicum, this time we're going to use your existing `program` folder instead of a separate one for this assignment, as the point of this practicum will be to set up `make` to work with your existing code.

So, start off by navigating to your checkout – `main_copy`, assuming you followed practicum 2 to the letter – do an "`svn update .`" command, and then go to your `program` directory.

If you do not have a main function defined anywhere, take this opportunity to make one quickly – create a `main.cpp`, and just define a standard `main` function, just so the whole program will link and create an executable!

So, first, compile everything, and make note of your command to do so, which will probably be something like:

```
g++ *.cpp -o prog1
```

Easy enough, right? Ok, now let's get to automating.

`make`, due to its age, has an unusual convention – instead of taking a specific file passed to it as an argument, whenever it runs, it always looks for a file called "`makefile`". So we're going to have to create a `makefile` for it.

Fire up your preferred text editor on your multilab account, and create a file named `makefile`. In it, put:

```
prog1:
	g++ *.cpp -o prog1
```

Note: the initial `prog1` should be directly against the left margin of the file, and the command under it *must* have a tab in front of it. Not a bunch of spaces, it specifically has to be a tab. Why? Because make is old, and at the time it was thought to be a good idea.

Now, save the file, and run `make`. What you should see is the command entered above immediately run for you. What's going on here is you're declaring a target for `make` – `prog1`, in this case – and then, since it's the *first* target in the `makefile`, typing `make` without any arguments will cause the command to be executed for you.

This, while it saves you a few keystrokes, is not all that useful. So we'll keep going.

First, let's separate out the command a bit. We should all be aware of the `-c` command line option for `g++` -- which compiles to an object file instead of an executable. Run:

```
g++ -c *.cpp
```

And then look at the object files that now exist in your directory. Next, we'll use `g++`'s built in linker to link these object files directly into an executable:

```
g++ *.o -o prog1
```

Which should give you an executable identical to the original one. Yes, there's a point to this, just wait… You might have noticed that the full compilation time on the multilab can take a bit. Since compiling all those files is what takes up most of that time, we're going to set up `make` so that it will only compile the .cpp files that have actually changed, and then link them all together. To do this is going to require a few steps.

First, `make` supports variables that can be declared once and then used in commands. We'll start by listing all of the object files we need to link the final executable as one of those variables. Open back up the `makefile`, and edit it as follows:

```
OBJECTS = Entity.o Item.o (listing all of your object files here,
separated by spaces)



prog1: $(OBJECTS)
	g++ $^ -o prog1
```

Now, this contains a few new features of `make`:

- Variables; declared in all caps, and followed by an = sign and then what's in the variable

- Use of a variable in a command – `OBJECTS` is the variable name, `$(OBJECTS)` is `make` syntax for reusing that variable.
- Dependency: Everything after the colon of the target line is a dependency of that target; we'll see how to use that a bit better shortly, but note that this is telling make that to build `prog1`, everything in `OBJECTS` has to be present.
- The `$^` special symbol – make replaces this symbol with the dependencies of the current target.

So, make these changes, drop back out of your editor, run `make`, and you should see it run `g++` with all of your object files and generate your executable.

Ok, so this works fine as long as the object files are already there; but we want make to automate *that*, too, so we've got a little bit more work to do.

Whenever you ask `make` to build something for you, it first checks to make sure its dependencies are there, and if they're not (or out of date – we'll talk about that in a second), it builds those dependencies for you – *if* it has rules for them. So what we're going to do next is write a `makefile` rule that will build our object files for us:

Add, to the bottom of your makefile:

```
%.o: %.cpp
	g++ -c $< -o $@
```

Now, we've got a few more concepts to add here:

- The `%` character – is a wildcard like * in most places; it just means anything can be used there. Note that once it's matched in the target, the same is used in the dependency. What this is telling `make` is that to build any target that ends in .o, it has a dependency of the same filename ending in .cpp, and then to execute the `g++` command below.
- The `$@` special symbol – `make` replaces this with the target name itself (which is especially useful here since we're using a wildcard and don't necessarily know what should be here until the wildcard is matched!).
- The `$<` special symbol – `make` replaces this with the first dependency. Why `$<` instead of `$^` used above? We'll see in a second when we add the next step…

So, now, you should be able to remove all of your object files and then run `make`, and watch it compile all of the object files for you and then link them all to the final program.

Now, for the next step – remove just one of your object files, and run `make` again. What you'll see this time is `make` building only the missing object file, and then relinking everything to produce your executable.

Now, one more trick. Edit one of your .cpp files, make a change, save it, quit back out, and run `make`: what you should see is it rebuilding that object file and then relinking. What's going on is `make` detects changes in dependencies, and if it sees that the output of a target is out of date, it rebuilds it.

But right now, it's only checking for changes in the .cpp file itself – changing the header files associated can cause differences, and we need to rebuild in that case. To do this in a simple way, we'll add another variable:

```
HEADERS = Entity.h Item.h (and the rest of your headers)
```

And then change the rule for compiling the object files to use these as a dependency:

```
%.o: %.cpp $(HEADERS)
	g++ -c $< -o $@
```

(This being where the use of `$<` instead of `$^` matters, as we just added more dependencies!)

Now, for one more addition:

At the bottom of your `makefile`, add one more target, `clean`:

```
clean:
	rm -f *.o prog1
```

Exit out, type "`make clean`", and it cleans up all of your temporary files. Then run `make` again, and rebuild them all.

And that's it for today.

Requirements

Just a functional `makefile` for your programming assignment, committed to your `program` directory.

This `makefile` will be required for your PA1 Part II submission as well!