What is `make`?

`make` is a system utility for managing the build process (compilation/linking/etc).

There are various versions of `make`; these notes discuss the GNU `make` utility included on Linux systems.

As the GNU Make manual* says:

> The `make` utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.

Using `make` yields a number of benefits, including:

- faster builds for large systems, since only modules that must be recompiled will be
- the ability to provide a simple way to distribute build instructions for a project
- the ability to provide automated cleanup instructions

`*http://www.gnu.org/software/make/manual/make.pdf`

Minimal Approach: Source Base

The following presentation is based upon the following collection of C source files:

`driver.c`	the main “driver”
`CSet.h`	the "public" interface of the CSet type
`CSet.c`	the implementation of the CSet type
`gradeCSet.h`	the "public" interface of the test harness
`gradeCSet.c`	the implementation of the test harness

The example is derived from an assignment that is occasionally used in CS 2506.

Minimal Approach: a simple makefile

Here's a minimal makefile for the given source base:

```
# CSet minimal makefile
#
SHELL=/bin/bash
#
# Specify compiler and compiler switches:
CC=gcc
CFLAGS=-std=c11 -Wall -W -O0 -ggdb3
#
# Build executable for testing:
driver: driver.c CSet.c gradeCSet.c
	$(CC) $(CFLAGS) -o driver driver.c CSet.c gradeCSet.c
#
# Remove object files:
clean:
	rm -f *.o driver
#
# Archive source and makefile:
package:
	tar cvf CSetCode.tar *.c *.h makefile
```

Minimal Approach: make options

The given makefile provides:

- a way to create an executable from the given source files: `make driver`

```
. . .
#
# Build executable for testing:
driver: driver.c CSet.c gradeCSet.c
	$(CC) $(CFLAGS) -o driver driver.c CSet.c gradeCSet.c
. . .
```

- a way to clear the directory of stale files: `make clean`

```
#
# Remove object files:
clean:
	rm -f *.o driver
```

- a way to package the source files: `make package`

```
#
# Archive source and makefile:
package:
	tar cvf CSetCode.tar *.c *.h makefile
```

Minimal Approach: Limitations

The given makefile does not take advantage of the most interesting feature of `make`:

- the ability to only recompile files that are affected by changes

Standard Example: Source Base

The following presentation is based upon the following collection of C modules:

`c05driver.c`	driver for testing code
`arrayList.h`	public interface for arrayList data structure
`arrayList.c`	implementation of arrayList functions
`MLBPerson.h`	public interface of MLBPerson data type
`MLBPerson.c`	implementation of MLBPerson functions
`mlbSelector.h`	public interface of mlbSelector tools
`mlbSelector.c`	implementation of mlbSelector functions
`alTester.h`	public interface of high-level testing tools
`alTester.c`	implementation of high-level testing functions
`alTestHelper.h`	public interface of low-level testing tools
`alTestHelper.c`	implementation of low-level testing tools

The example is derived from an assignment that has been used in CS 2505.

Determining Dependencies

`grep` can be used to discover **`include`** directives related to files in the project:

We ignore **`include`** directives that load Standard Library headers.

We must pay attention to **`include`** directives for both .h and .c files in each module.

```
alTester.h:  #include "arrayList.h"
alTester.c:  #include "alTester.h"
alTester.c:  #include "alTestHelper.h"
alTester.c:  #include "MLBPerson.h"
alTester.c:  #include "mlbSelector.h"
```

From the information above, we see that the `alTester` module depends on:

- the `alTesterHelper` module
- the `MLBPerson` module
- The `mlbSelector` module

Determining Dependencies

For the other modules, we get these **include** directives:

```
alTestHelper.h:  #include "arrayList.h"
alTestHelper.c:  #include "alTestHelper.h"
alTestHelper.c:  #include "MLBPerson.h"
```

```
c05driver.c:  #include "mlbSelector.h"
c05driver.c:  #include "arrayList.h"
c05driver.c:  #include "MLBPerson.h"
c05driver.c:  #include "alTester.h
```

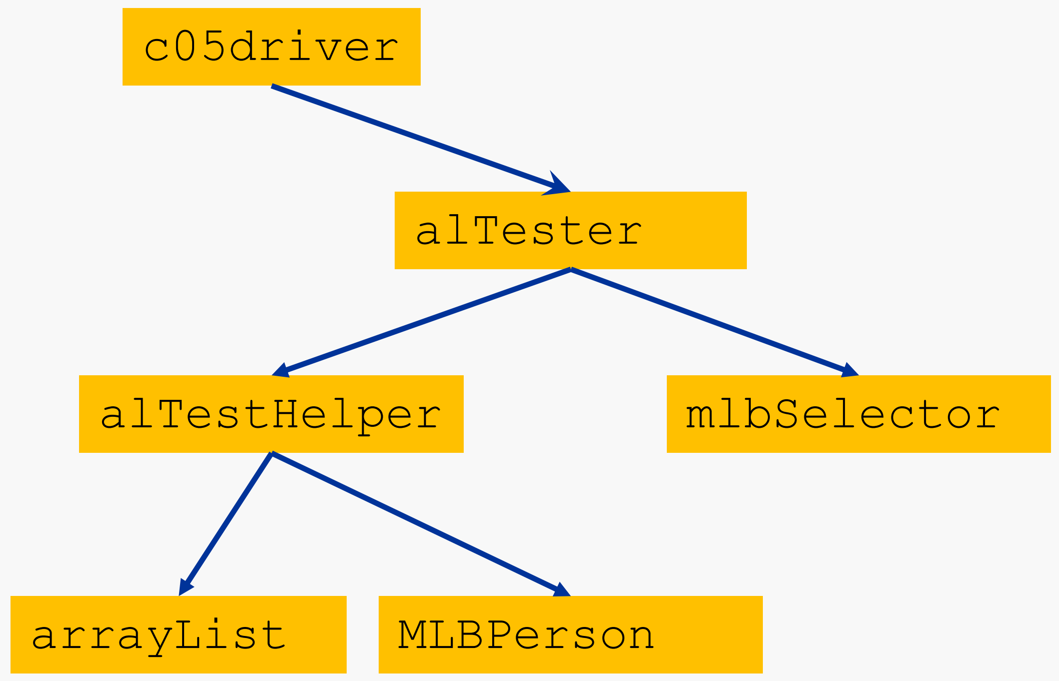
So, the `alTestHelper` module depends on:

- the `arrayList` module
- the `MLBPerson` module

And, the `c05driver` module depends on all the others (as we might expect).

Dependency Map

The C modules exhibit the following dependencies (due to **`include`** directives):


$$\begin{tikzcd}
\texttt{c05driver} \arrow[d] & & \\
\texttt{alTester} \arrow[d] \arrow[dr] & & \\
\texttt{alTestHelper} \arrow[d] \arrow[dr] & \texttt{mlbSelector} & \\
\texttt{arrayList} & \texttt{MLBPerson} &
\end{tikzcd}$$

A module must be recompiled/relinked if any module it depends on, directly or indirectly, has been changed.

Makefiles and Rules

You use a kind of script called a *makefile* to tell `make` what to do.

A simple makefile is just a list of rules of the form:

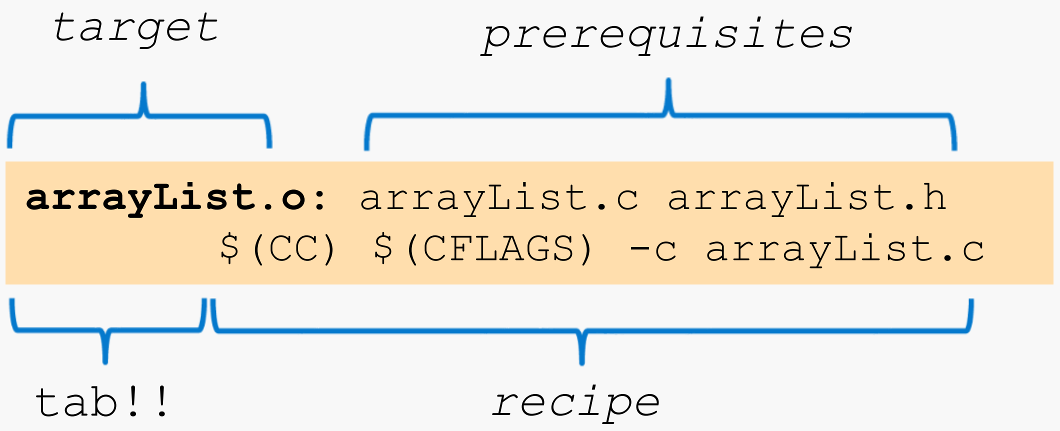
```
target … : prerequisites …
        recipe
        …
```

Prerequisites are the files that are used as input to create the target.

A *recipe* specifies an action that make carries out.

Defining a Simple Rule

Here is a simple rule for compiling `arrayList.c` (and so producing `arrayList.o`):



```
arrayList.o: arrayList.c arrayList.h
	$(CC) $(CFLAGS) -c arrayList.c
```

So, if we invoke `make` on this rule, `make` will execute the command:

```
gcc -std=c11 -Wall -W -ggdb3 -c arrayList.c
```

which will (ideally) result in the creation of the object file `arrayList.o`.

Defining a Simple Rule

Here is a simple rule for compiling `arrayList.c` (and so producing `arrayList.o`):

```
arrayList.o: arrayList.c arrayList.h
        $(CC) $(CFLAGS) -c arrayList.c
```

The list of prerequisites guarantees that if `arrayList.c` (or `arrayList.h`) changes, then `arrayList.o` will be recreated to reflect changes that may have affected it.

We could invoke this rule as follows:

```
centos > make arrayList.o
gcc -std=c11 -Wall -W -ggdb3 -c arrayList.c
```

Invoked again, make detects no need to recompile:

```
centos > make arrayList.o
make: 'arrayList.o' is up to date.
```

Defining a More Complex Rule

Here is a simple rule for producing `alTestHelper.o`:

```
alTestHelper.o: alTestHelper.c alTestHelper.h arrayList.o MLBPerson.o
        $(CC) $(CFLAGS) -c alTestHelper.c
```

Now, `alTestHelper.c` will be recompiled if any of these conditions hold:

- any prerequisite is more recent than `alTestHelper.o`
- any prerequisite has a prerequisite that is more recent than itself (in which case that prerequisite will also be recompiled)

```
centos > make alTestHelper.o
gcc -std=c11 -Wall -W -ggdb3 -c arrayList.c
gcc -std=c11 -Wall -W -ggdb3 -c MLBPerson.c
gcc -std=c11 -Wall -W -ggdb3 -c alTestHelper.c
```

```
centos > touch MLBPerson.o
centos > make alTestHelper.o
gcc -std=c11 -Wall -W -ggdb3 -c alTestHelper.c
```

Using Targets as Prequisites

Note that in the rule just given we have specified other targets as prerequisites:

```
alTestHelper.o: alTestHelper.c alTestHelper.h arrayList.o MLBPerson.o
        $(CC) $(CFLAGS) -c alTestHelper.c
```

That’s what enables the “chaining” effect seen below:

```
centos > touch MLBPerson.c
centos > make alTestHelper.o
gcc -std=c11 -Wall -W -ggdb3 -c MLBPerson.c
gcc -std=c11 -Wall -W -ggdb3 -c alTestHelper.c
```

Makefile Variables

We can define variables in our makefile and use them in recipes:

```
CC=gcc
CFLAGS=-std=c11 -Wall -W
```

```
arrayList.o:  arrayList.c arrayList.h
    $(CC) $(CFLAGS) -c arrayList.c
```

This would make it easier to alter the compiler options for all targets (or to change compilers).

Syntax note: no spaces around `'='`.

Rules Without Prerequisites

We can also define a rule with no prerequisites; the most common use is probably to define a cleanup rule:

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

Invoking `make` on this target would cause the removal of all object and stackdump files from the directory.

This rule is handy for backing up the current source files:

```
package:
    tar cvf c05_source.tar *.h *.c
```

A Complete Makefile

Here is a complete makefile for the example project:

```
# Specify shell to execute recipes
SHELL=/usr/bin/bash

# Set compilation options:
#
#    -std=c11    use C11 Standard features
#    -Wall       show "all" warnings
#    -W          show even more warnings (annoyingly informative)
#
# Specify compiler and compiler switches:
CC=gcc
CFLAGS=-std=c11 -Wall -W


. . .
```

A Complete Makefile

```
. . .
#
# Rule for making a debug build:
debug: c05driver.c alTester.o
    $(CC) $(CFLAGS) -o c05 -O0 -ggdb3 c05driver.c alTester.o \
                 alTestHelper.o mlbSelector.o arrayList.o MLBPerson.o

#
# Rule for making a release build:
release: c05driver.c alTester.o
    $(CC) $(CFLAGS) -o c05 c05driver.c alTester.o alTestHelper.o \
                                  mlbSelector.o arrayList.o MLBPerson.o
. . .
```

A Complete Makefile

```
. . .

# Rules for building the modules:
alTester.o: alTester.c alTester.h alTestHelper.o mlbSelector.o
	$(CC) $(CFLAGS) -c alTester.c

alTestHelper.o: alTestHelper.c alTestHelper.h arrayList.o MLBPerson.o
	$(CC) $(CFLAGS) -c alTestHelper.c

mlbSelector.o: mlbSelector.c mlbSelector.h
	$(CC) $(CFLAGS) -c mlbSelector.c

arrayList.o: arrayList.c arrayList.h
	$(CC) $(CFLAGS) -c arrayList.c

MLBPerson.o: MLBPerson.c MLBPerson.h
	$(CC) $(CFLAGS) -c MLBPerson.c
. . .
```

A Complete Makefile

```
. . .

#
# Rule for packing up source files:
package:
	tar cvf c05_source.tar *.h *.c

#
# Rule for cleaning object files from directory:
clean:
	rm -f *.o c05
```

Running `make`

`make` can be invoked in several ways, including:

```
make
make <target>
make -f <makefile name> <target>
```

In the first two cases, `make` looks for a makefile, in the current directory, with a default name. GNU `make` looks for the following names, in this order:

```
GNUmakefile
makefile
Makefile
```

If no target is specified, `make` will process the first rule in the makefile.

Examples using `make`

Using the makefile shown above, and the source files indicated earlier:

```
centos > ll
total 60
-rw-rw-r--. 1 wmcquain wmcquain 5502 Sep 20 20:56 alTester.c
-rw-rw-r--. 1 wmcquain wmcquain 1939 Sep 20 20:56 alTester.h
-rw-rw-r--. 1 wmcquain wmcquain 5823 Sep 20 20:56 alTestHelper.c
-rw-rw-r--. 1 wmcquain wmcquain 3720 Sep 20 20:56 alTestHelper.h
-rw-rw-r--. 1 wmcquain wmcquain 3563 Sep 20 20:56 arrayList.c
-rw-rw-r--. 1 wmcquain wmcquain 5072 Sep 20 20:56 arrayList.h
-rw-rw-r--. 1 wmcquain wmcquain 3846 Sep 20 20:56 c05driver.c
-rw-rw-r--. 1 wmcquain wmcquain 1117 Sep 20 21:02 makefile
-rw-rw-r--. 1 wmcquain wmcquain 2224 Sep 20 20:56 MLBPerson.c
-rw-rw-r--. 1 wmcquain wmcquain 1706 Sep 20 20:56 MLBPerson.h
-rw-rw-r--. 1 wmcquain wmcquain 1139 Sep 20 20:56 mlbSelector.c
-rw-rw-r--. 1 wmcquain wmcquain  697 Sep 20 20:56 mlbSelector.h

centos > make release
gcc -std=c11 -Wall -W -c arrayList.c
gcc -std=c11 -Wall -W -c MLBPerson.c
gcc -std=c11 -Wall -W -c alTestHelper.c
gcc -std=c11 -Wall -W -c mlbSelector.c
gcc -std=c11 -Wall -W -c alTester.c
gcc -std=c11 -Wall -W -o c05 c05driver.c alTester.o alTestHelper.o \
                                    mlbSelector.o arrayList.o MLBPerson.o
```

Examples using make

Now, I'll modify one of the C files and run make again:

```
centos > touch MLBPerson.c

centos > make release
gcc -std=c11 -Wall -W -c MLBPerson.c
gcc -std=c11 -Wall -W -c alTestHelper.c
gcc -std=c11 -Wall -W -c alTester.c
gcc -std=c11 -Wall -W -o c05 c05driver.c alTester.o alTestHelper.o \
                           mlbSelector.o arrayList.o MLBPerson.o
```

The only recipes that were invoked were those for the targets that depend on MLBPerson.c.