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`

Source Base

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

`pqtest.c`	the main "driver"
`TestPlace.h`	test harness for the `Place` type
`TestPlace.c`	
`TestPlaceQueue.h`	test harness for the `PlaceQueue` type
`TestPlaceQueue.c`	
`MarkUp.h`	point annotation code used in automated testing
`MarkUp.c`	
`PlaceUtilities.h`	display code for `Place` and `PlaceQueue` objects
`PlaceUtilities.c`	
`PlaceQueueUtilities.h`	
`PlaceQueueUtilities.c`	
`Place.h`	structured type encapsulating GIS information
`Place.c`	
`PlaceQueue.h`	interface for managing a queue of `Place` objects
`PlaceQueue.c`	
`Queue.h`	generic queue implementation
`Queue.c`	

The example is derived from a programming assignment used in CS 2506 during Fall 2013.

Dependencies

The C source files use the following `include` directives related to files in the project:

```
pqtest.h:
  TestPlace.h
  TestPlaceQueue.h
```

```
TestPlace.h:
  Place.h
TestPlace.c:
  PlaceUtilities.h
  MarkUp.h
```

```
TestPlaceQueue.c:
PlaceQueueUtilities.h
PlaceUtilities.h
MarkUp.h
```

```
PlaceUtilities.h:
  Place.h
```

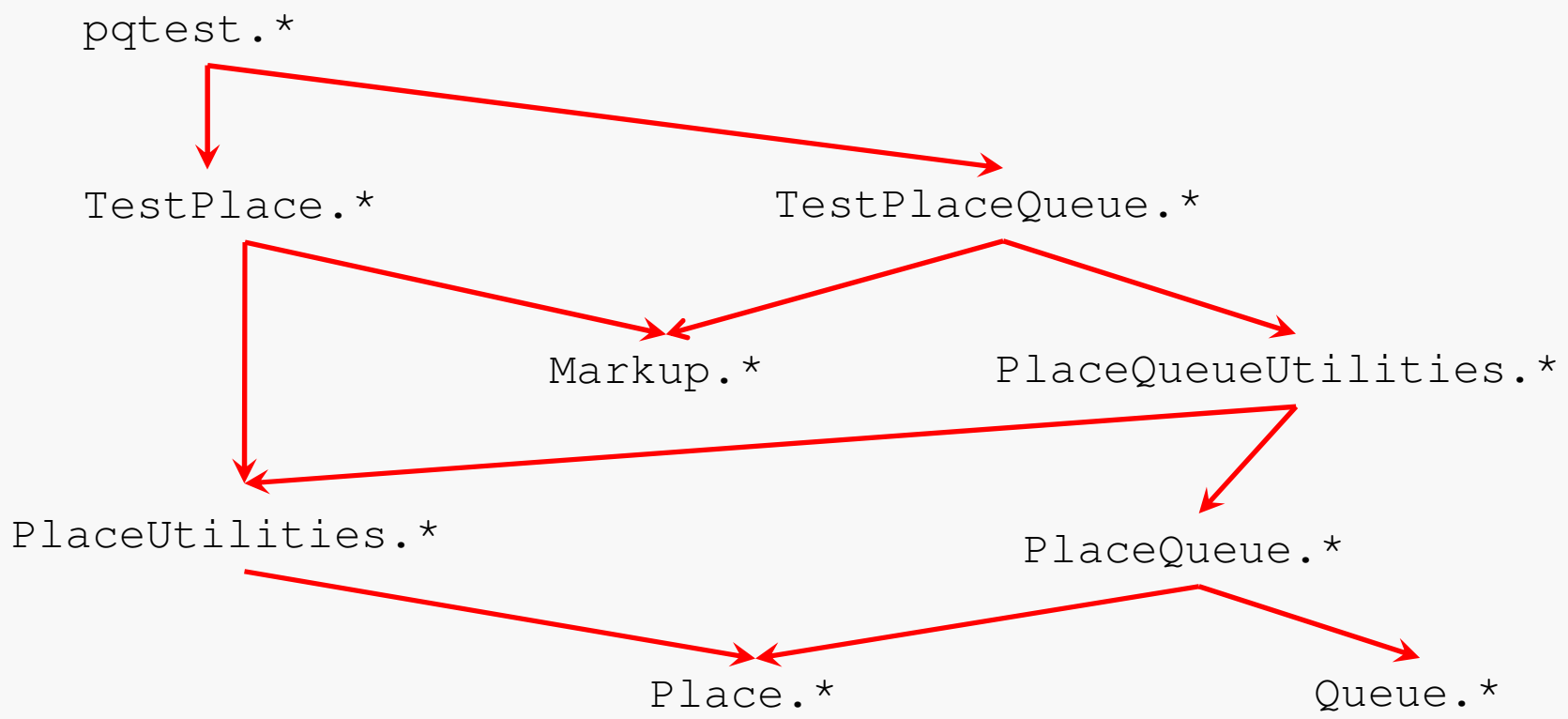
```
PlaceQueue.h:
  Place.h
  Queue.h
```

```
PlaceQueueUtilities.h:
  PlaceQueue.h
PlaceQueueUtilities.c:
  PlaceUtilities.h
  MarkUp.h
```

We need to understand how the inclusions affect compilation…

Dependency Map

The C source files exhibit the following dependencies (due to `include` directives):



We need to understand the dependencies in order to define the *rules* `make` will apply.

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:

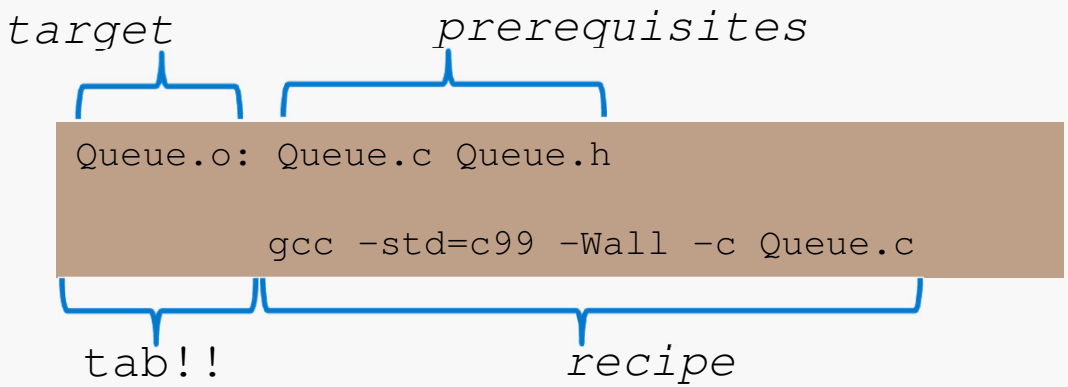
target … : *prequisites* …
recipe
…

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 `Queue.c` (and so producing `Queue.o`):



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

```
gcc -std=c99 -Wall -c Queue.c
```

Defining a More Complex Rule

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

```
TestPlace.o: TestPlace.c TestPlace.h \
                 MarkUp.c MarkUp.h \
                 PlaceUtilities.c PlaceUtilities.h \
                 Place.c Place.h

    gcc -std=c99 -c TestPlace.c MarkUp.c PlaceUtilities.c Place.c
```

Now, we have some issues:

- This doesn't save us any rebuilding… every C file that `TestPlace.o` depends on will be recompiled every time we invoke the rule for that target.
- There is a lot of redundancy in the statement of the rule… too much typing!
- What if we wanted to build for debugging? We'd need to add something (for instance, `-ggdb3`) to the recipe in every rule. That's inefficient.

Using the Dependencies

We can specify targets as prerequisites, as well as C source files:

```
TestPlace.o: TestPlace.c TestPlace.h PlaceUtilities.o MarkUp.o
    gcc -std=c99 -c TestPlace.c

PlaceUtilities.o: PlaceUtilities.c PlaceUtilities.h Place.o
     gcc -std=c99 -c PlaceUtilities.c

MarkUp.o: MarkUp.c MarkUp.h
     gcc -std=c99 -c MarkUp.c
```

Now, if we invoke make on the target `TestPlaceQueue.o`:

- make examines the modification time for each direct and indirect prerequisite for `TestPlace.o`
- each involved target is rebuilt, by invoking its recipe, iff that target has a prerequisite, that has changed since that target was last built

Makefile Variables

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

```
CC=gcc
CFLAGS=-O0 -m32 -std=c99 -Wall -W -ggdb3
```

```
TestPlace.o: TestPlace.c TestPlace.h PlaceUtilities.o MarkUp.o
    $(CC) $(CFLAGS) -c TestPlace.c
```

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

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.

A Complete Makefile

Here is a complete makefile for the example project:

```
# Makefile for assignment C3, CS 2506, Fall 2013
#
SHELL=/bin/bash

# Set compilation options:
#
#   -O0         no optimizations; remove after debugging
#   -m32        create 32-bit executable
#   -std=c99    use C99 Standard features
#   -Wall       show "all" warnings
#   -W          show even more warnings (annoying)
#   -ggdb3      add extra debug info; remove after debugging
#
CC=gcc
CFLAGS=-O0 -m32 -std=c99 -Wall -W -ggdb3
OBJECTS = pqtest.o Place.o PlaceQueue.o Queue.o TestPlace.o \
          TestPlaceQueue.o MarkUp.o PlaceUtilities.o \
          PlaceQueueUtilities.o

# Build the test code (full project):

pqtest: $(OBJECTS)
    $(CC) $(CFLAGS) -o pqtest $(OBJECTS)
...
```

A Complete Makefile


```
...
# Rules for components:
pqtest.o: pqtest.c TestPlace.o TestPlaceQueue.o
    $(CC) $(CFLAGS) -c pqtest.c

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

PlaceQueue.o: PlaceQueue.c PlaceQueue.h Place.o Queue.o
    $(CC) $(CFLAGS) -c PlaceQueue.c

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

TestPlace.o: TestPlace.c TestPlace.h PlaceUtilities.o MarkUp.o
    $(CC) $(CFLAGS) -c TestPlace.c

TestPlaceQueue.o: TestPlaceQueue.c TestPlaceQueue.h \
                  PlaceQueueUtilities.o MarkUp.o
    $(CC) $(CFLAGS) -c TestPlaceQueue.c

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

A Complete Makefile


```
...

PlaceUtilities.o: PlaceUtilities.c PlaceUtilities.h Place.o
    $(CC) $(CFLAGS) -c PlaceUtilities.c

PlaceQueueUtilities.o: PlaceQueueUtilities.c \
                       PlaceQueueUtilities.h \
                       PlaceQueue.o PlaceUtilities.o
    $(CC) $(CFLAGS) -c PlaceQueueUtilities.c

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

# Cleaning rules:
clean:
    rm -f *.o *.stackdump

cleantext:
    rm -f *.txt

cleanallfiles:
    rm -f *.o *.txt
```

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:

```
[wdm@VMCentos64 Make]$ make pqtest
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c Place.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c PlaceUtilities.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c MarkUp.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c TestPlace.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c Queue.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c PlaceQueue.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c PlaceQueueUtilities.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c TestPlaceQueue.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c pqtest.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -o pqtest pqtest.o Place.o
PlaceQueue.o Queue.o TestPlace.o TestPlaceQueue.o MarkUp.o
PlaceUtilities.o PlaceQueueUtilities.o
[wdm@VMCentos64 Make]$
```

Since I hadn't compiled anything yet, `make` invoked all of the rules in `Makefile`.

Examples using `make`

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

```
[wdm@VMCentos64 Make]$ touch MarkUp.c

[wdm@VMCentos64 Make]$ make pqtest
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c MarkUp.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c TestPlace.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c TestPlaceQueue.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c pqtest.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -o pqtest pqtest.o Place.o
PlaceQueue.o Queue.o TestPlace.o TestPlaceQueue.o MarkUp.o
PlaceUtilities.o PlaceQueueUtilities.o
[wdm@VMCentos64 Make]$
```

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

Examples using `make`

Now, I'll modify a "deeper" C file and run `make` again:

```
[wdm@VMCentos64 Make]$ touch Place.c

[wdm@VMCentos64 Make]$ make pqtest
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c Place.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c PlaceUtilities.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c TestPlace.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c PlaceQueue.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c PlaceQueueUtilities.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c TestPlaceQueue.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c pqtest.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -o pqtest pqtest.o Place.o
PlaceQueue.o Queue.o TestPlace.o TestPlaceQueue.o MarkUp.o
PlaceUtilities.o PlaceQueueUtilities.o
[wdm@VMCentos64 Make]$
```

Again, the only files that were recompiled were the ones depending on the changed file.

Examples using `make`

Of course, we can also build "secondary" targets:

```
[wdm@VMCentOS64 Make]$ make PlaceQueue.o
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c Place.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c Queue.c
gcc -O0 -m32 -std=c99 -Wall -W -ggdb3 -c PlaceQueue.c
[wdm@VMCentOS64 Make]$
```

The only files that were compiled were the ones on which the specified target depends.

Using Implicit Rules

Finally, we can simplify the makefile by taking advantage of the fact that make applies certain implicit rules by default:

```
...
# Build components:
pqtest.o:                Place.o TestPlace.o TestPlaceQueue.o
Place.o:                 Place.h
PlaceQueue.o:            PlaceQueue.h Place.o Queue.o
Queue.o:                 Queue.h
TestPlace.o:             TestPlace.h PlaceUtilities.o MarkUp.o
TestPlaceQueue.o:        TestPlaceQueue.h PlaceQueueUtilities.o MarkUp.o
PlaceUtilities.o:        PlaceUtilities.h Place.o
PlaceQueueUtilities.o: PlaceQueueUtilities.h PlaceQueue.o PlaceUtilities.o
MarkUp.o:                MarkUp.h
...
```

See the GNU Make manual for a full discussion of this.