Make Facility

* The make facility is one of the most important tools that you will learn in this course. You will use it in many of your remaining computing science courses
* **make** will automatically construct the executable form of a program from a description of the source files and the dependencies between the files
* Once you have produced a Makefile for a program you don't need to remember how to compile it, or even the files you have changed. **make** handles these details automatically for you
* The main input to **make** is a Makefile (or makefile). It describes how to compile and link the program
* The Makefile is divided into two main parts:
  1. macro definitions
  2. dependency declarations

Macro Definitions

* Macros are used extensively in UNIX and C programming, get used to them
* A macro is a text replacement facility
* A macro has a name, which is a character string, and a body, which is also a character string, wherever the name occurs in the file it is replaced by the body of the macro

* In **make** we use macros to describe the compiler we are using, the compiler options, and the list of files we need. It allows us to parameterize the Makefile
* A <u>macro definition</u> has the following format:

```
macro-name = macro-body
```

* A macro definition can appear anywhere in a Makefile
* In **make**, a macro is invoked by mentioning its name, preceded by a $, if the macro is more than one character long the name must be placed in parenthesis
* In our example makefile we have

```
OBJECTS = main.o phone.o
```

* When we use $(OBJECTS) in the rest of the Makefile it is replaced by the two file names, as we add more files to the program we only need to change the line where OBJECTS is declared, the rest of the Makefile stays the same - this reduces mistakes!!
* Similarly we have the lines

```
CC = gcc
CFLAGS = -Wall -ansi
```

* The first line specifies the C compiler that we use, and the second line defines the compiler flags. Again we can quickly change the compiler options without scanning through the entire Makefile for occurrences.

Dependency Declarations

* Make has a set of default rules that know about Unix's file naming conventions, we use suffixes to indicate the type of file, start with a dot (**.**) and have one or more letters, the common ones are:

name**.c**	a C program
name**.h**	a header file
name**.o**	an object file
name**.s**	an assembler file

* Make uses the file suffixes to determine how proceed. If make needs a **.o** file and can find a **.c** file with the same prefix, it knows that it can use the C compiler to produce the **.o** file
* For example, if name.o is required i.e. mentioned on the right side of a dependency rule, but there is no rule for name.o then **make** will look for a file named name.c, and if it finds one will run the C compiler on it, the use of **default rules** reduces the typing you must do *(but I avoid them)** The dependency declarations tell **make** which files depend on which other files
* In the case of our example, phone (the executable for our program) depends on both main.o and phone.o; we indicate this in the following way:

```
phone :main.o phone.o
```

* The file named on the left of the **:** depends on the files listed on the right
* Similarly, both main.o and phone.o depend on phone.h There are two ways that we could specify this

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

or (though this may be too obscure)

```
%.o     :  %.c phone.h
           $(CC) $(CFLAGS) -c $<
```

* After each dependency declaration we can list the commands that construct the dependent files, these commands must be indented, there must be a tab at the beginning of the line.

* So in order to create phone, we use the following specification to link the object modules together.

```
phone : main.o phone.o
        $(CC) $(CFLAGS) -o phone main.o phone.o
```

* When we need to re-create phone, **make** will automatically execute the **gcc** command
* When is a dependency declaration used?

* When make is invoked it looks at the time of last modification of all the files mentioned in the Makefile. If the last time of modification of a file on the right side of a dependency declaration is more recent than one of the files on the left side then the commands are executed!
* In our example, if either main.o or phone.o is more recent than phone, the gcc command will be executed, otherwise **make** knows that phone is up-to-date and does nothing
* How does **make** know how to make main.o and phone.o, there are no commands associated with their dependencies?

* If **make** is started without arguments, it will use the first dependency declaration as its target. That is, it will make the file on the left side of the first dependency rule in the Makefile
* In our example Makefile, **phone** is the first file mentioned, so by default make will execute that command first

* You can specify the target when you call make. If we enter the command

make main.o

* then make will only run the C compiler on main.c to produce main.o, it will <u>not produce a new version of phone</u>
* The **most common** file that you produce should be in the **first** dependency declaration, so you don't need to mention it each time you use make
* Besides making the executable of a program there are other standard things that are placed in a Makefile, some of these operations include <u>removing</u> temporary files, <u>installing</u> the executable in a standard place, running standard test cases, and <u>printing</u> the program

* We could add the following rules to our Makefile:

```
clean:  $(OBJECTS)
        rm $(OBJECTS)

install :  phone
           cp phone   $HOME/bin/phone
```

* When we execute the command

make clean

All the .o files for the phone program will be deleted.