

Makefile I

- Makefiles are useful when we want to break our program into different files
- it is always a good idea to split your program into many *coherent* files
- in general we would deal with two types of files:
 - *source files*: these are usually kept in a directory called `src`
 - * example: `.c` or `.C` or `.cpp` files
 - *include files*: these are usually kept in a directory called `include`, although some people tend to keep them in the `src` directory, its your choice, really!
 - * example: `.h` or `.H` or `.hpp` files

Makefile II

- the build process has two parts:
 - the compiler takes the source files and outputs object files which have a `.o` extension
 - the linker takes the object files, links them to system libraries, e.g. `math.h` and creates an executable which by default is called `a.out` but you could name it whatever you want

Makefile III

- basic make command syntax:

```
target: dependencies-line1 \  
        dependencies-line2 \  
        . many-more-dependencies-lines \  
        .  
[tab] system-command
```

- the [tab] is really really really important!
- the [tab] has to be a [tab] and not whitespace
- the “\” is the line-continuation mark
- the spaces in the dependencies-lines doesn't matter, the weird spacing above is deliberate!

Makefile IV

- example:

- suppose we have two files `prog1.H` and `prog1.C` and would like to create an executable called `prog1` from these two

- consider:

```
prog1:          prog1.o
               g++ -g -Wall -O -o prog1 prog1.o
```

```
prog1.o:        prog1.C \
                prog1.H
               g++ -g -Wall -O -c prog1.C -o prog1.o
```

- can we point out the target, dependencies and the system-command bits in the above?

Makefile V

- tricks to make your makefile writing easier:

- use *variables*:

```
CC = g++
OPT = -g -Wall -O
INCLUDES =
CCFLAGS = $(OPT) $(INCLUDES)
OBJECTS = prog1.o
```

- use makefile *idioms*:

```
prog1:          $(OBJECTS)
               $(CC) $(CCFLAGS) -o prog1 $(OBJECTS)
```

```
prog1.o:       prog1.C \
               prog1.H
               $(CC) $(CCFLAGS) -c $< -o $@
```

- * \$(FOO) fetches the value of the variable FOO: avoids retyping/cutting-n-pasting long expressions
- * \$< means the first dependency and \$@ means the target: avoids retyping/cutting-n-pasting dependency, target names and hence errors

Makefile VI

- using a makefile:
 - suppose we save the following in a file called prog1Makefile

```
CC = g++
OPT = -g -Wall -O
INCLUDES =
CCFLAGS = $(OPT) $(INCLUDES)
OBJECTS = prog1.o
prog1:      $(OBJECTS)
            $(CC) $(CCFLAGS) -o prog1 $(OBJECTS)

prog1.o:   prog1.C \
            prog1.H
            $(CC) $(CCFLAGS) -c $< -o $@
```

- then we use: `make -f prog1Makefile`

Makefile VII

- the .PHONY targets:
- these are used to declare targets which are not file-names, they produce on-demand-use tools

```
.PHONY: clean  
clean:  
    -rm prog1 $(OBJECTS)
```

- append the above lines to the prog1Makefile file
- use of .PHONY targets: `make clean -f prog1Makefile`
- this gets rid of all .o files and the executable which are usually huge files

Makefile VIII

- you could have mutiple .PHONY targets:

```
.PHONY: cleanall cleanobj
```

```
cleanobj:  
    -rm $(OBJECTS)
```

```
cleanall: cleanobj  
    -rm prog1
```