Advanced Software Techniques

More about makefiles

So far ...

You know

a little

about

makefiles

But

there's

more

makefile macros

makefiles

can have

macros

They're a bit different from #define in C

They're more like

variables

that are used like

constants

Remember "magic numbers"?

Syntax

Macro name followed by = followed by definition

Usually upper-case

Spaces around = depends (OK for nmake)

Use it:

```
$(macroName)
```

Example

```
CC = cl
LD = cl
CFLAGS = /c
LDFLAGS = /Fe
OBJ = main.obj input.obj output.obj
STD_HEADERS = prototypes.h
EXE = application.exe
```

```
$(EXE): $(OBJ)
    $(LD) $(OBJ) $(LDFLAGS)$(EXE)

main.obj: main.c $(STD_HEADERS)
    $(CC) main.c $(CFLAGS)

input.obj: input.c $(STD_HEADERS) special.h
    $(CC) input.c $(CFLAGS)

output.obj: output.c $(STD_HEADERS) special.h
    $(CC) output.c $(CFLAGS)
```

OK, so why?

Using macros
makes it easier
to make changes
later

For example:

- you change compilers
- you add another standard header file
- command line options change for your compiler

Aside

CC?
LD?

In UNIX, the compiler is CC
(short for C Compiler)
and
the linker is called LD
(short for loader)

And some of the others ...

OBJ contains the list of all of the object files

CFLAGS contains the

C compiler

command-line options

LDFLAGS contains the linker command-line options

How about a generic makefile!

```
CC = cl
LD = cl
CFLAGS = /c
LDFLAGS = /Fe
SRC = main.c input.c output.c
OBJ = $(SRC: .c=.obj)
STD_HEADERS = prototypes.h
EXE = application.exe
```

```
all: $(SRC)   $(EXE)

$(EXE): $(OBJ)
    $(LD) $(OBJ) $(LDFLAGS)$@

input.obj: input.c $(STD_HEADERS) special.h

output.obj: output.c $(STD_HEADERS) special.h

main.obj: main.c $(STD_HEADERS)

.c.obj:
    $(CC) $< $(CFLAGS)
```

We specify generic rules
(if they apply)

We specify specific dependencies
(if we need to)

The first dependency
is a general one
saying:

"If you're making,
it depends on the
executable
and
source files."

And it's always the FIRST dependency that is used (unless we tell it otherwise)

What's with those weird things?!?!?

$@

What???

$<

Double what???

$@

represents

the name of the

file to be made

`$<`

represents

the name of the

related file

that caused the action

Also:

$*

represents the prefix shared by target and dependent files

$?

represents

the names

of the changed dependents

Let's go back ...

I said:

"And it's always the FIRST

dependency that

is used

(unless we tell it otherwise)"

That implies:

we can tell it otherwise

We can put at the end of the makefile:

```
clean:
	rm *.obj
	rm *.exe
```

If we then execute NMAKE with:

NMAKE clean

It'll find the "clean" dependency

and

execute the rules

under it

In this case,

it gets rid of

.obj and .exe files

We do this if we want to rebuild the entire project

This doesn't do the rebuild ...

It only sets it up

So, if we do
NMAKE clean
followed by
NMAKE,

it'll rebuild
the project

We can add other artificial dependency/rule pairs

For example,
to create documentation

Or to

work with

revision control systems

Or to

archive files in a .ZIP file

Alternative makefiles

The default makefile name is simply:

makefile

If you want to specify a different makefile name,

use the -f command-line option

Example

```
nmake -f makefile.mak
```

Last, but not least ...

Comments in a makefile

start with #

on the extreme left

Example

```
# prepare for complete rebuilding
clean:
	rm *.obj
	rm *.exe
```

Final caveat

NMAKE and the Linux/UNIX MAKE commands can have sneaky differences

And ...

Microsoft is notorious
for changing syntax
but the Internet
doesn't necessarily change with
it

For further reference:

http://viralpatel.net/taj/tutorial/makefile-tutorial.php is a pretty good tutorial on the UNIX-based make http://www.tidytutorials.com/2009/08/nmake-makefile-tutorial-and-example.html is a small tutorial on a different use of NMAKE http://msdn.microsoft.com/en-us/library/dd9y37ha.aspx is the Microsoft documentation for NMAKE (but it's hard to navigate)