**The Problem**

As usual here's a simple Makefile that illustrates the use of VPATH and vpath.

SRCS := foo.c bar.c baz.c

OBJS := $(SRCS:.c=.o)

.PHONY: all

all: $(OBJS)

foo.o: foo\_header.h

bar.o: string.h

Assume that the source files foo.c, bar.c and baz.c are all in the same directory as the Makefile.  When you run GNU Make you get an error:

$ make

cc    -c -o foo.o foo.c

make: \*\*\* No rule to make target `string.h', needed by `bar.o'.  Stop.

because string.h (which is a common header file) cannot be found.  There are a couple of solutions to this:

1. Change string.h to the full path where it is actually located.  For example, on my machine, its full path is /usr/include/string.h.

2. Use VPATH or vpath to find it.

The first solution is actually what happens if you use any sort of automatic dependency generation (likemakedepend).  It will create dependencies containing the full path of #included header files.  But we could also use VPATH or vpath.

**VPATH and vpath**

The VPATH is a list of directories to be searched for missing source files (actually for missing prerequisites: they don't have to be source files, but VPATH and vpath are best only used for source files).

The list can be separated by spaces or colons (and on Windows versions of GNU Make it's possible to use semicolon as a list element separator): the best approach is probably to use spaces as this works across all platforms.

When GNU Make cannot find a prerequisite it will search the directories in the VPATH list (from first to last) and stop at the first directory in which it finds the missing prerequisite.   It then substitutes the location where the missing prerequisite is found for the name specified in the Makefile.

So, we can fix the broken Makefile above by specifying:

VPATH = /usr/include

anywhere in the Makefile.  Now a GNU Make works correctly:

$ make

cc    -c -o foo.o foo.c

cc    -c -o bar.o bar.c

cc    -c -o baz.o baz.c

If you want to clear the VPATH then simply set it to empty:

VPATH =

The problem with this approach is that VPATH is a hammer to crack a nut: it would have been better to tell GNU Make just where to find string.h.  Using this VPATH solution means that any time GNU Make can't find a prerequisite it'll go searching down the  VPATH.   That can be a maintenance problem:

1. If there are files with the same name in different directories, VPATH could end up picking the wrong one.

2. The list of directories must be maintained in the correct order, and this order may need to be translated for other applications (e.g. for the compiler's #include search path).  vpath enables you to minimize these problems by using a smaller hammer!  The vpath directive has both a search path and a pattern.  Only missing prerequisites matching the pattern are searched using the associated path.  So vpath makes it possible, for example, to specify just a path to search for header ( .h) files:

vpath %.h /usr/include

The % is the wildcard and matches anything, so that vpath directive solves the problem of the missingstring.h in the example Makefile.

The vpath syntax is a little more complicated than VPATH and has three forms:

1. vpath pattern path This sets the search path (colon or blank separated) for the pattern.

2. vpath pattern This clears the path for the specified pattern.

3. vpath Clears all vpath settings

Neither VPATH nor vpath were designed to find system headers (like string.h); they were