1. What is a makefile?

Make is a program that looks for a file called *"makefile"* or *"Makefile"*, within the makefile are variables and things called dependencies. There are many things you can do with makefiles, if all you've ever done with makefiles is compile C or C++ then you are missing out.

Pretty much anything that needs to be compiled (postscript, java, Fortran), can utilize makefiles.

2. Why are makefiles useful?

Makefiles can make the compilation procedure much **faster**.

- The compilation is done using a single command
- Only the files that must be compiled are compiled
- Allows managing large programs with a lot of dependencies.

3. A makefile example

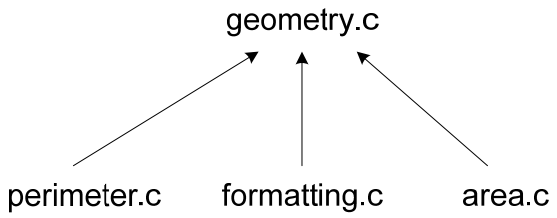
```
#
#Makefile for the geometry program
#
area.o:             area.h area.c
                    gcc -c area.c
perimeter.o:        perimeter.h perimeter.c
                    gcc -c perimeter.c

formatting.o:       formatting.h formatting.c
                    gcc -c formatting.c

geometry:           area.o perimeter.o formatting.o geometry.c
                    gcc -o geometry area.o perimeter.o formatting.o geometry.c
```

Some explanations

The program we are compiling is called **geometry**, and its code is in the file **geometry.c**. Dependencies exist as shown in the following figure.



So to procure the executable program geometry we need to compile as:

```
gcc -o geometry area.o perimeter.o formatting.o geometry.c
```

To produce each object file we have to compile each one of them separately, that is

```
gcc -c area.c
gcc -c perimeter.c
gcc -c formatting.c
```

So the steps for producing the executable program “geometry” are:

```
gcc -c area.c
gcc -c perimeter.c
gcc -c formatting.c
gcc -o geometry area.o perimeter.o formatting.o geometry.c
```

If we use the makefile given above the compilation command will be just one line:

```
make geometry
```

Let’s study each part of this makefile.

```
1   #Makefile for the geometry program
2
3   area.o:       area.h area.c
4                 gcc -c area.c
5
6   perimeter.o:  perimeter.h perimeter.c
7                 gcc -c perimeter.c
8
9   formatting.o: formatting.h formatting.c
10                gcc -c formatting.c
11
12  geometry:     area.o perimeter.o formatting.o geometry.c
13                gcc -o geometry area.o perimeter.o formatting.o geometry.c
```

Line 1	Comments
Lines 2 – 5 – 8 – 11	Just empty space, nothing functional
Lines 3 – 4	Rule area.o, produces object file area.o
Lines 6 – 7	Rule perimeter.o, produces object file perimeter.o
Lines 9 – 10	Rule formatting.o, produces object file formatting.o
Lines 12 – 13	Rule geometry, produces the executable file geometry

4. Parts of a makefile

4.1 Comments

Comments in a makefile start with character #

4.2 Dependences – Rules – Actions

A rule is defines the action to be taken from the make command. In the example above we have 4 rules, area, perimeter, formatting and geometry.

```
ruleName:       file1 file2
                action command
```

ruleName	Is the name of the rule
file1 file2	These files are the depends of the rule. That is the action defined by the rule will be taken only if the depends of the rule are available. Here the action defined by rule area will be taken only if files file1 and file2 are available. If they are not available the rule area will search for a rule that produces each one of these files and call it. If it fails finding a rule even for one file it exits giving an error message
action command	It is the action of rule ruleName

How rules work

Lets assume that our directory consist of the files

- area.c
- area.h
- perimeter.c
- perimeter.h
- formatting.c
- formatting.h
- geometry.c

and that we execute the command

```
make geometry
```

Then the rule geometry will be triggered.

```
geometry:   area.o perimeter.o formatting.o geometry.c
            gcc -o geometry area.o perimeter.o formatting.o geometry.c
```

The first depend of the geometry rule is the file area.o. So make will search to find the rule that produces file area.o. This rule is the rule area.o. So command

```
gcc –c area.c
```

will be executed

The next depend is file perimeter.o. This file doesn't exist either. Make will find and execute rule perimeter.o. So command

```
gcc –c perimeter.c
```

will be executed

The next depend is file formatting.o and so rule formatting.o will be executed. So command

```
gcc –c formatting.c
```

will be executed

As file geometry.c is available in the current directory rule geometry is finally going to be executed.

```
gcc -o geometry area.o perimeter.o formatting.o geometry.c
```

The output to the screen will be:

```
$ make geometry
gcc -c area.c
gcc -c perimeter.c
gcc -c formatting.c
gcc -o geometry area.o perimeter.o formatting.o geometry.c
```

If for example the file perimeter.c was not present in the directory then rule perimeter would fail.

```
$ make geometry
gcc -c area.c
make: *** No rule to make target ‘perimeter.c’, needed by ‘perimeter.o’.  Stop.
```

4.3 Timestamps

Make uses timestamps to locate the files that have been modified since the last time the make was executed. If none of the files of our example were modified and make is called again the result will be:

```
$ make geometry
make: ‘geometry’ is up to date.
```

Now we modify only file perimeter.c and execute again the make command. Make will execute the actions of all rules that are depended from file perimeter.c. These rules are perimeter.o and geometry (geometry uses perimeter.o so it is implicitly affected by perimeter.c)

```
$ make geometry
gcc -c perimeter.c
gcc -o geometry area.o perimeter.o formatting.o geometry.c
```

4.4 Macros

Macros are used to give names to variables. It is something equivalent to defining constants in C programs. The makefile we are studying uses gcc as a compiler in all action lines. If we wanted to change the compiler then we must change it to all the four lines it appears. A better approach is to use a macro.

```
#
#Makefile for the geometry program
#
COMPILER=gcc
area.o:           area.h area.c
                  $(COMPILER) -c area.c

perimeter.o:      perimeter.h perimeter.c
                  $(COMPILER) -c perimeter.c

formatting.o:     formatting.h formatting.c
                  $(COMPILER) -c formatting.c

geometry:   area.o perimeter.o formatting.o geometry.c
            $(COMPILER) -o geometry area.o perimeter.o formatting.o geometry.c
```

A macro is defined in as

```
Macro_Variable = macro_Name
```

The macro variable is “called” by using

```
$(Macro_Variable)
```

5. Finally some syntax

Rules

Be careful when defining rules, the action line must begin with a tab, otherwise you will get the error message

```
*****missing separator. Stop
```

```
ruleName:  file1 file2
    tab    command
```

Single lines

The dependences of a rule must be placed in a continued line. The same stands for the actions of the rule. If you want to separate one line you must use character \

So writing

```
First_Part_Of_Line_1  Second_Part_Of_Line_1
```

Is the same as

```
First_Part_Of_Line_1  \
Second_Part_Of_Line_1
```

6. Include a clean rule

It is very useful to include a "Clean Rule" in your makefile. You can use this to remove all:

- Obsolete files
- All backup files
- Core files
- … …

This will allow you to remove all files that are no more needed or can be produced from your code files by compiling. This also allows better compression as in this way you do not include in your compressed file files that you can reproduce.

An example of a clean rule is the following.

```
clean:
        -$(RM) *.o *.obj *.exe core *~ MAKE.log Makefile.bak $(PROGS)
```

Here:

RM	Is a macro that was previously defined and corresponds to the delete command (rm most of the times)
PROGS	Is a macro that holds the names of all the binaries produced by the project. For our example this would be geometry

The macro definitions would be

```
RM    = rm –f            # rm –f delete without asking for verification
PROGS = geometry
```

7. Inference Rules

Inference rules generalize the build process so you don't have to give an explicit rule for each target. As an example, compiling C source (.c files) into object files (.obj files) is a common occurrence. Rather than requiring a statement that each .obj file depends on a like-named .c file, Make uses an inference rule to infer that dependency. The source determined by an inference rule is called the "*inferred source*".

Inference rules are rules distinguished by the use of the character **'%'** in the dependency line. The **'%'** (rule character) is a **wild card**, **matching zero or more characters**. As an example, here is an inference rule for building .obj files from .c files:

```
%.obj:  %.c
        $(CC) $(CFLAGS) –c $(.SOURCE)
```

This rule states that a .obj file can be built from a corresponding .c file with the shell line "$(CC) $(CFLAGS) –c $(.SOURCE)". The .c and .obj files share the same root of the file name.

When the source and target have the same file name except for their extensions, this rule can be specified in an alternative way:

```
.c.obj :
        $(CC) $(CFLAGS) –c $(.SOURCE)
```

Make predefines the "`%.obj : %.c`" inference rule as listed above so the example we have been working on now becomes much simpler:

```
OBJS = main.obj io.obj
CC = bcc
MODEL = s
CFLAGS = –m$(MODEL)
project.exe : $(OBJS)
        tlink c0$(MODEL) $(OBJS), $(.TARGET),, c$(MODEL) /Lf:\bc\lib
$(OBJS) : incl.h
```