CSD101: Introduction to computing and programming (ICP)

How does GCC work?

- Different compilers may work slightly differently. But GCC has the following work flow:

  C program $\overset{Pre-processor}{\rightarrow}$ Augmented C program $\overset{Compiler}{\rightarrow}$ Assembly code $\overset{Assembler}{\rightarrow}$ Object code .o file $\overset{Linker}{\rightarrow}$ Executable.

- Using switches GCC can be made to stop at any stage to get the corresponding output. For example,
  `gcc -E <file.c>` -o fileAug.c
  will run just the pre-processor and output the augmented C program in `fileAug.c`. The switch `-S` will produce assembly code.
- In addition GCC has a large number of switches that allow control over different stages of the work flow. See documentation at `https://gcc.gnu.org/`

The `make` utility I

- Large programs are normally distributed over many files where each file tends to implement one coherent piece of functionality. An example is the standard C library.
- When changes are made to one or more files generating an executable by typing in commands at the command line can be extremely difficult and error prone. It may be almost impossible when the number of files is very large.
- The `make` utility allows us to specify file dependencies and actions so that changes to one or more files can trigger an orderly sequence of compilations to generate the final executable. The utility can be used in more general settings where a sequence of actions follow due to changes in some files that initiate actions on other files that depend on them.

The `make` utility II

- For example, in C an executable is created from a set of `.o` files which in turn are created by compiling source code files. If a change is made in the source code of a library then all files that use functions from that library will have to be compiled to `.o` files and then linked to get the executable.

makefile

- By default, the `make` program reads its specifications from a file called `makefile` or `Makefile`.
- Various versions of `make` are available on different platforms and they are mostly compatible with each other for simple makefiles. However, most versions also have their own extensions that will not work on other platforms.
- The `make` we discuss is GNU `make` that is available on Linux and MAC systems. It is used to build executables for the Linux kernel, GCC compiler, Firefox browser, LibreOffice system amongst others.

Make specification I

- A make specification in a `makefile` contains a sequence of *rules* having the following structure.
- Structure of a rule:

```
target ... [target1 ...] : pre-requisites
TAB command1
TAB command2
TAB ...
```

- `target` is the executable file that must be built. The pre-requsite contains file names of files on which `target` depends.
- `command n` lines are actions that must be done to build the `target`. **Note** the TAB character in front of each command line. That is necessary. Omitting it leads to errors.

Make specification II

- A rule can contain just a `target` without any pre-requisites. Such a target is called a *phony* target. Phony targets are meant to just execute a sequence of commands and their name(s) must be passed in as arguments to the `make` command.
- By default, the `make` command without arguments will build the first target in the `makefile`. This can trigger a chain of other commands because the pre-requisites of `target` may itself be a target.

Example[3] - version 1

```
edit : main.o kbd.o command.o display.o insert.o \
        search.o files.o utils.o
    cc -o edit main.o kbd.o command.o display.o insert.o \
        search.o files.o utils.o

main.o : main.c defs.h
    cc -c main.c

kbd.o : kbd.c defs.h command.h
    cc -c kbd.c

command.o : command.c defs.h command.h
    cc -c command.c

display.o : display.c defs.h buffer.h
    cc -c display.c
```

[3] make manual

Example contd.

```
insert.o : insert.c defs.h buffer.h
    cc -c insert.c

search.o : search.c defs.h buffer.h
    cc -c search.c

files.o : files.c defs.h buffer.h command.h
    cc -c files.c

utils.o : utils.c defs.h
    cc -c utils.c

clean :
    rm edit main.o kbd.o command.o display.o insert.o \
        search.o files.o utils.o
```

Example version 2, use of variables

```
objs = main.o kbd.o command.o display.o insert.o \
        search.o files.o utils.o

edit : $(objs)
    cc -o edit $(objs)

main.o : main.c defs.h
    cc -c main.c

kbd.o : kbd.c defs.h command.h
    cc -c kbd.c

command.o : command.c defs.h command.h
    cc -c command.c

display.o : display.c defs.h buffer.h
    cc -c display.c
```

Example version 2 contd.

```
insert.o : insert.c defs.h buffer.h
    cc -c insert.c

search.o : search.c defs.h buffer.h
    cc -c search.c

files.o : files.c defs.h buffer.h command.h
    cc -c files.c

utils.o : utils.c defs.h
    cc -c utils.c

clean :
    rm edit $(objs)
```

Example version 3, implicit rules

```
objs = main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
edit : $(objs)
    cc -o edit $(objs)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
    rm edit $(objs)
```

Alternate style - using implicit rules

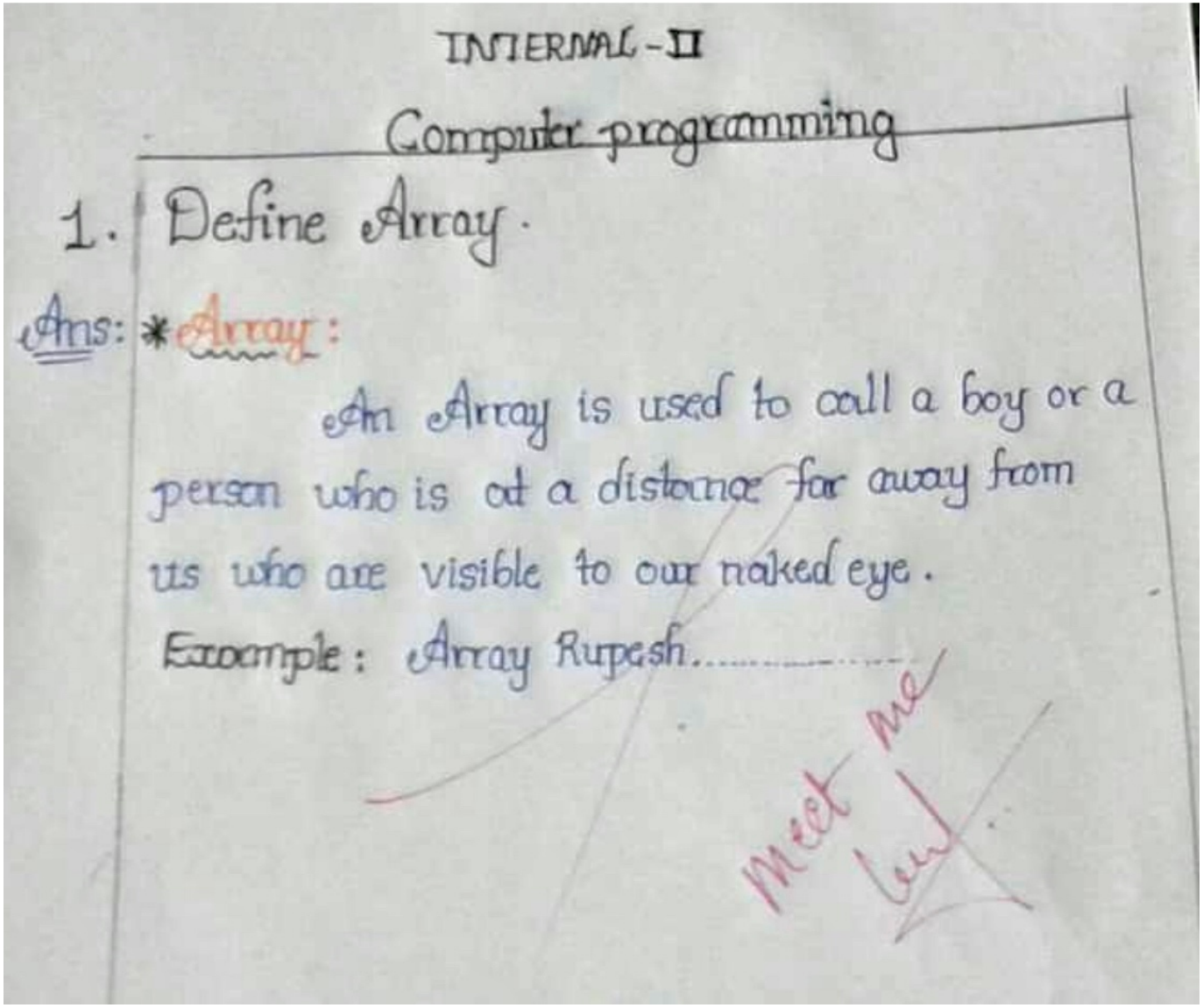
```
objects = main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
edit : $(objects)
    cc -o edit $(objects)
$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

Makes the *makefile* more compact but it is a matter of taste which style of *makefile* is used.

Further information

- GNU `make` has many more features. We have discussed the basic features that will allow us to build executables easily when multiple files are involved.
- For further details see the `make` manual available at: `https://www.gnu.org/software/make/manual/`.

Hope you do better than this. Bonne chance



INTERNAL - II

Computer programming

1. Define Array.

Ans: * Array :

An Array is used to call a boy or a person who is at a distance far away from us who are visible to our naked eye.

Example : Array Rupesh.....

meet me