HEADERS

Purpose

It is efficient to divide the project to modules, and reuse code so that it is made sure that its not written twice .

As programs grow larger and larger (and include more files), it becomes increasingly tedious to have to forward declare every function you want to use that lives in a different file.

Writing Header file

main.c

```
#include<stdio.h>
int square(int x);        //forward declaration
int main()
{
int x=square(2);
printf("%d",x);
}
```

square.c

```
int square(int x)
{return x*x;}
```

Above code changes to :

main.c

```
#include<stdio.h>
#include "square.h"                    //int square(int x);
int main()
{
square(2);
}
```

square.c

```
int square(int x)
{return x*x;}
```

square.h

```
// This is start of the header guard.SQUARE_H can be any unique name.  By convention, we use the name of the header file.

#ifndef SQUARE_H
#define SQUARE_H
// This is the content of the .h file, which is where the declarations go
int square(int x); // function prototype for square.h don't forget the semicolon!
// This is the end of the header guard
#endif
```

Header consists of two parts

- header guards
- declarations

Header Guards

when same header files are included in multiple source file, the definitions will be declared multiple times which may result in error. For solving this problem ,header guard is used.When header guard is used , a macro-variable is defined when header file is included for first time.

```
#ifndef SQUARE_H
#define SQUARE_H
……..

#endif
```

Declarations

forward function declarations

compiling

gcc -I. main.c square.c -lm -o pgm

-I is to specify Include directory where header files are stored

-I. specify compiler to search current directory for header files

MakeFile

Makefiles are special format files that together with the *make* utility will help you to automatically build and manage your projects.

Structure of Makefile

target1: dependencies1

[tab] system command1

target2: dependencies2

[tab] system command2

Command

make target1

When make is invoked,
it check whether all dependicies are present. If not they are recursively generatoed
it detect all changes made to dependencies
For targets which are having their dependent files altered are regenerated . All *targets* which where dependent on generated target are recompiled recursively.
This technique saves time used for recompiling whole system.

Eg.

Makefile

```
all: hello

hello: main.o factorial.o hello.o
    g++ main.o factorial.o hello.o -o hello

main.o: main.cpp
    g++ -c main.cpp

factorial.o: factorial.cpp
    g++ -c factorial.cpp

hello.o: hello.cpp
    g++ -c hello.cpp

clean:
    rm *o hello
```

user/Desktop$ ***make hello***