Separate Header and Source Files

Purpose: divide the project into modules, and reuse code.

As programs grow larger and larger (and include more files), it becomes increasingly tedious to forward declare every function you want to use that lives in a different file. A smaple code with forward declation is shown below

main.c

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

square.c

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

Above code can be rewritten as three separte files :

main.c

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

square.c

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

square.h

```
//Starts with a header guard. Any unique name can be used
//instead of SQUARE_H. By convention, we use the name of the
//header file.
#ifndef SQUARE_H
#define SQUARE_H
// what follows is the content of the .h file, which is where the
// declarations go
int square(int x); // function prototype for square.h. Do not
                   // forget the semicolon!
#endif   //end of the
     //header gaurd
```

A Header file consists of two parts

- header guard
- declarations

Header Guard

when same header files are included in multiple source file, the declarations will occur multiple times. For solving this problem, header guard is used. This defines a macro-variable the first time the file is included. Any further inclusions are prevented by having the content of the header file within the guard.

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

#endif
```

Declarations

Forward function declarations.

Compiling

gcc –I. main.c square.c –lm –o pgm

–I specify Include directory where header files are stored

–I. search current directory for header files

Note: A .c file without main can be compiled using the partial compilation option -c

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

When the command `make` is invoked, it checks whether all dependicies are present. If not, they are recursively generated. It detects all changes made to dependencies. Targets which are having their dependent files altered are regenerated . All *targets* which were dependent on generated targets are recompiled recursively. This technique saves time by compiling only those files which are modified.

A sample 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
```

Document prepared by: Arjun Suresh.
Comments / Suggestions to 123arjunsuresh@gmail.com