**Why Makefile require?**

* When we want execute 2-3 files together we use statement as below :

g++ main.cpp message.cpp -o output

* But when there are number of files needs to be executed above mentioned command cannot be useful that is why we use Make file.
* Compiling the source code files can be tiring, especially when you have to include several source files and type the compiling command every time you need to compile. Makefiles are the solution to simplify this task.
* Makefiles are special format files that help build and manage the projects automatically.
* For example, let’s assume we have the following source files.
* main.cpp
* message.cpp
* message.h

**message.h**

The following is the code for message.h header file −

class message

{

private:

/\* data \*/

public:

void PrintMessage();

};

**message.cpp**

The following is the code for message.cpp source file −

#include<iostream>

#include "message.h"

using namespace std ;

void message::PrintMessage()

{

cout<<"Makefile example\n";

}

**main.cpp**

The following is the code for main.cpp source file −

#include "message.h"

#include <iostream>

using namespace std ;

int main()

{

message m;

m.PrintMessage();

return 0;

}

* The trivial way to compile the files and obtain an executable, is by running the command −

g++ main.cpp message.cpp -o output

* In this example we have only 3 files and we know the sequence of the function calls. Hence, it is feasible to type the above command and prepare a final executable file.
* However, for a large project where we have thousands of source code files, it becomes difficult to maintain the builds.
* The **make** command allows you to manage large programs or groups of programs. As you begin to write large programs, you notice that re-compiling large programs takes longer time than re-compiling short programs. Moreover, you notice that you usually only work on a small section of the program ( such as a single function ), and much of the remaining program is unchanged.
* For above example only we will execute using Makefile that has below:

# target : depedencies

# action

all: main.o message.o

g++ main.o message.o -o output

main.o: main.cpp

g++ -c main.cpp

message.o: message.cpp message.h

g++ -c message.cpp

clean:

rm \*.o output

* When all file needs to be execute we call as : make all

or any of the files changed next time when executes : make all changed one

only will executes

**Variables**

One feature of makefile is use of MACRO variables how to use it we will see it

below :

# target : depedencies

# action

CC=g++

CFLAGS=-c

all: main.o message.o

$(CC) main.o message.o -o output

main.o: main.cpp

$(CC) $(CFLAGS) main.cpp

message.o: message.cpp message.h

$(CC) $(CFLAGS) message.cpp

clean:

rm \*.o output

**Multiline variables**

* These type of variables make makefile code more redundant and easy to understand by expanded multiline Variables for example as below mentioned :

# target : depedencies

# action

CC=g++

CFLAGS=$(CC)-c

all: main.o message.o

$(CC) main.o message.o -o output

main.o: main.cpp

$(CFLAGS) main.cpp

message.o: message.cpp message.h

$(CFLAGS) message.cpp

clean:

rm \*.o output

**Simply expanded Variables**

* These type of variables make makefile code more redundant and easy to understand by expanded Simply expanded Variables to use these type of variables we need to use ‘:=’ or :’::=’ for example as below mentioned :

# target : depedencies

# action

CC :=g++

CFLAGS :=$(CC)-c

CC :=g++ -c

it will be expanded as

CFLAGS :=g++ -c

CC := g++ -c

**Using implicit rules**

* whenever we remove action part for compiling to generate object file make file explicitly consider to compile respective file with to generate .o files

**Automatic variable**

* There are number of Automatic variables following are few mentioned with examples

1)$@ : replaced by name of target

2)$< : replaced by name of first dependency file

3)$^ : replaced by name of all dependency files

# target : depedencies

# action

CC=g++

CFLAGS=-c

all: main.o message.o

$(CC) $^-o $@

main.o: main.cpp

$(CC) $(CFLAGS) $<

message.o: message.cpp message.h

$(CC) $(CFLAGS) $<

clean:

rm \*.o output

**PatternRules**

* Instead of righting each and every commands for every files use %.o as target and %.cpp or %.c as dependency for all files and perform actions

accordingly.

CC=g++

CFLAGS=-c

all: main.o message.o

$(CC) $^ -o $@

%.o: %.cpp %.h

$(CC) $(CFLAGS) $^

clean:

rm \*.o all