A Short Introduction to Makefile

Make Utility and Makefile

- The *make* utility is a software tool for managing and maintaining computer programs consisting many component files. The *make* utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.
- *Make* reads its instruction from Makefile (called the descriptor file) by default.
- Makefile sets a set of rules to determine which parts of a program need to be recompile, and issues command to recompile them.
- Makefile is a way of automating software building procedure and other complex tasks with dependencies.
- Makefile contains: **dependency rules**, **macros** and **suffix**(or **implicit**) **rules**.

```
/* main.cpp */
#include <iostream>
#include "functions.h“

using namespace std;
int main()
{
   print_hello();
   cout << endl;
   cout << "The factorial of 5 is " <<
factorial(5) << endl;
   return 0;
}
```

```
/* hello.cpp */
#include <iostream>
#include "functions.h"

using namespace std;
void print_hello()
{
  cout << "Hello World!";
}
```

```
/* factorial.cpp */
#include "functions.h"

int factorial(int n)
{
   int i, fac = 1;
   if(n!=1){
      for(i=1; i<= n; i++)
         fac *= i;
      return fac;
   }
   else return 1;
}
```

```
/* functions.h */
#if !defined(_FUNC_H_)
#define _FUNC_H_

void print_hello();
int factorial(int n);

#endif   /* if !define(_FUNC_H_) */
```

Command Line Approach to Compile

- `g++ -c hello.cpp main.cpp factorial.cpp`
- `ls *.o`

```
factorial.o hello.o main.o
```

- `g++  -o prog factorial.o hello.o main.o`
- `./ prog`

```
Hello World!
The factorial of 5 is 120
```

- Suppose we later modified hello.cpp, we need to:
- `g++ -c hello.cpp`
- `g++  -o prog factorial.o hello.o main.o`

Example Makefile

```
# This is a comment line
CC=g++
# CFLAGS will be the options passed to the compiler.
CFLAGS= -c  -Wall

all: prog

prog: main.o factorial.o hello.o
	$(CC) main.o factorial.o hello.o -o prog

main.o: main.cpp
	$(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
	$(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
	$(CC) $(CFLAGS) hello.cpp

clean:
	rm -rf   *.o
```

Basic Makefile Structure

Dependency rules

- A rule consists of three parts, one or more targets, zero or more dependencies, and zero or more commands in the form:

  target: dependencies

  <tab> commands to make **target**

  - <tab> character MUST NOT be replaced be spaces.
  - A "**target**" is usually the name of a file(e.g. executable or object files). It can also be the name of an action (e.g. clean)
  - "dependencies" are files that are used as input to create the **target**.
  - Each "command" in a rule is interpreted by a shell to be executed.
  - By default, *make* uses /bin/sh shell.
  - Typing "make **target**" will:
    1. Make sure all the dependencies are up to date
    2. If target is older than any dependency, recreate it using the specified commands.

- By default, typing “make” creates first target in Makefile.
- Since prog depends on main.o factorial.o hello.o, all of object files must exist and be up-to-date. *make* will check for them and recreating them if necessary

- Phony targets
  - A phony target is one that isn't really the name of a file. It will only have a list of commands and no dependencies.

E.g.
```
clean:
	rm -rf  *.o
```

Macros

- By using macros, we can avoid repeating text entries and makefile is easy to modify.
- Macro definitions have the form:

  NAME = text string

  e.g. we have: CC=g++

- Macros are referred to by placing the name in either parentheses or curly braces and preceding it with $ sign.
  - E.g. $(CC) main.o factorial.o hello.o -o prog

Internal macros

- Internal macros are predefined in *make*.
- "*make -p*" to display a listing of all the macros, suffix rules and targets in effect for the current build.

Special macros

- The macro @ evaluates to the name of the current target.
  - E.g.

```
prog1 : $(objs)
        $(CXX) -o $@ $(objs)
```

is equivalent to

```
prog1 : $(objs)
        $(CXX) -o prog1 $(objs)
```

Suffix rules

A way to define default rules or implicit rules that *make* can use to build a program. There are *double-suffix* and *single-suffix*.

- Suffix rules are obsolete and are supported for compatibility. Use pattern rules (a rule contains character '%') if possible.
- Doubles-suffix is defined by the source suffix and the target suffix . E.g.

```
.cpp.o:
        $(CC) $(CFLAGS)  -c  $<
```

  - This rule tells *make* that .o files are made from .cpp files.
  - $< is a special macro which in this case stands for a .cpp file that is used to produce a .o file.
- This is equivalent to the pattern rule "%.o : %.cpp"

```
%.o : %.cpp
        $(CC) $(CFLAGS)  -c  $<
```

Command line macros

- Macros can be defined on the command line.
  - E.g. make DEBUG_FLAG=-g

How Does Make Work?

- The *make* utility compares the modification time of the target file with the modification times of the dependency files. Any dependency file that has a more recent modification time than its target file forces the target file to be recreated.
- By default, the first target file is the one that is built. Other targets are checked only if they are dependencies for the first target.
- Except for the first target, the order of the targets does not matter. The make utility will build them in the order required.

A New Makefile

```
# This is a comment line
CC=g++
# CFLAGS will be the options passed to the compiler.
CFLAGS=-c –Wall
OBJECTS  = main.o hello.o factorial.o
all: prog

prog: $(OBJECTS)
	$(CC) $(OBJECTS)  -o prog

%.o:  %.cpp
	$(CC) $(CFLAGS)  $<

clean:
	rm -rf  *.o
```

- **Reference**

http://www.gnu.org/software/make/manual/html_node/