

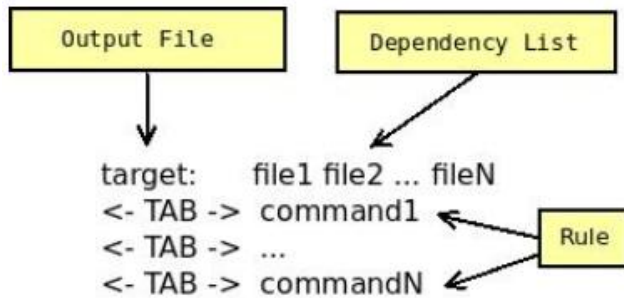
Exercise 2 : Makefiles - Compiling for host

Exercise 2.1 : Makefiles???

Writing makefiles is in itself writing in a scripting language meaning that one has to abide by the language rules that apply. Therefore before the first makefile is to be written certain keywords and concept must be known. Answer the following questions and remember to use code snippets if it serves to pave the way for better understanding:

- **What is a target?**

- The target of a makefile is the file to be created by the makefile, also called the output file.



The picture above is showing where the Target, Dependency list and the rules are located

- **What is a dependant and how is it related to a target?**

- After the output file and a colon (*target:*) comes the Dependency List. The target depends on the dependants.

- **Does it matter whether one uses tabs or spaces in a makefile?**

- Multiple target files and multiply source files must be separated by a space. To apply rules, ones must use Tab. Mixing up the two of them will result in a "Missing seperator. Stop" error output.

- **How do you define and use a variable in a makefile?**

- A variable is defined by using =
I.e. THIS = a b c
The variable "This" is now set to a b c, and you can reference it by using the dollar sign (\$) and braces({YourVariableNameHere}):
\${THIS}
A good practice is to always variable with capital letters.

- **Why use variables in a makefile?**

- A variable can represent a list of file names, which makes it a lot faster to call all of the files at once compared to typing all of the file names over and over.

- **How do you use a created makefile?**

- After creating a makefile, we could call it hello, you type in the command "make" to the terminal. If you type in make without any arguments, only the first rule in the file will be executed.

- **In the makefile scripting language they often refer to built-in variables such as these**

- **\$@, \$< and \$^ - explain what each of these represent.**
 - **\$@** This refers to the file name of the target of the rule. If the target is an archive member, the built-in variable refers to the name of the archive file.
 - **\$<** Refers the name of the first prerequisite (the first listed after the target).
 - **\$^** Refers to all the prerequisites.

- **\$(CC) and \$(CXX):**

- **What do they refer to?**

- **How do they differ from each other?**

- **\$(CC)** A recipe for compiling C programs, while **\$(CXX)** is a recipe for compiling C++ filer.

- **\$(CFLAGS) and \$(CXXFLAGS):**
 - **What do they refer to?**
 - **How do they differ from each other?**
 - The \$(CFLAGS) is a built in rule, that creates extra flags for det C compiler, while \$(CXXFLAGS) creates extra flags for the C++ compiler.
 - **What does \$(SOURCES : .cpp = .o) mean? Any spaces in this text???**
 - This commands that whatever object we put in front of it (ie. OBJECTS = \$(SOURCES:.cpp=.o)) is assigned the value of SOURCES, but with .o after it instead of .cpp.
You are not allowed to put any spaces in this command.
-

Exercise 2.2 : Using makefiles - starting out

Write a makefile for the program hello you created in the first exercise. Add a target all that compiles your program, furthermore use variables to specify the following:

- The name of the executable
- The used compiler.

Compile your program using make and execute it.

Add two targets to your makefile; clean that removes them all object files as well as the executable. Add a target help that prints a list of available targets.

Remember to verify via tests that all three targets do as expected.

```
SOURCES=hello.cpp

OBJECTS=${SOURCES:--cpp=.o}

EXECUTABLE=RUN

CXX=g++

all:${EXECUTABLE}

${EXECUTABLE}:${OBJECTS}
    ${CXX} -o ${EXECUTABLE} ${OBJECTS}

hello.o:hello.cpp
    ${CXX} -c hello.cpp

clean:
    rm -rf *.o && rm ${EXECUTABLE}    // Implemented after testing
                                     // and executing the first part
help:                                // of the program, beforehand
    @echo You can make:              // -
    @echo all                        // -
    @echo clean                      // -
    @echo help                       // -
```

The final code after implementing 'clean' and 'help'.

```

stud@stud-virtual-machine:~/maketest$ ls
hello.cpp  makefile
stud@stud-virtual-machine:~/maketest$ make -f makefile
g++ -o RUN hello.cpp
stud@stud-virtual-machine:~/maketest$ ./RUN
Hello world
FFS
stud@stud-virtual-machine:~/maketest$ make help
You can make:
all
clean
help
stud@stud-virtual-machine:~/maketest$ make clean
rm -rf *.o && rm RUN
stud@stud-virtual-machine:~/maketest$ ls
hello.cpp  makefile
stud@stud-virtual-machine:~/maketest$

```

Final test after implementing both 'clean' and 'help'

Exercise 2.3 : Program based on multiple files

Exercise 2.3.1 : Being explicit

Create a simple program parts consisting of 5 files:

*part1.cpp

contains 1 simple function part1() that prints "This is part 1!" on stdout

- part1.h
contains the definition of part1()
- part2.cpp
contains 1 simple function part2() that prints "This is part 2!" on stdout
- part2.h
contains the definition of part2()
- main.cpp
contains main() which calls part1() and part2()

Create a makefile for parts. As in Exercise 2.2 and specify the executable and the used compiler by means of variables. Add targets all, clean and help 2 as in Exercise 2.2.

The following is the source code for part1.h, part1.cpp, part2.h, part2.cpp and main.cpp:

```

//part1.hpp
#include <iostream>

class Part1{
public:
    void part1();
};

//part1.cpp
#include "part1.hpp"

void Part1::part1(){
    std::cout<<"This is part 1!" << std::endl;
};

//part2.hpp
#include <iostream>

class Part2{
public:
    void part2();
};

```

```
};

//part2.cpp
#include "part2.hpp"

void Part2::part2(){
    std::cout<<"This is part 2!" << std::endl;
};

//main.cpp
#include "part1.hpp"
#include "part2.hpp"

int main(){
    Part1 fisk;
    Part2 hund;
    fisk.part1();
    hund.part2();

    return 0;
}
```

Source code for the parts-makefile

The following is the makefile for parts. The dependencies are the ones created in the source code above.

```
SOURCE=main.cpp part1.cpp part1.hpp part2.cpp part2.hpp

OBJECTS=${SOURCES:-cpp=.o}

EXECUTABLE=RUN

CXX=g++

all:${EXECUTABLE}

${EXECUTABLE}:${OBJECTS}
    ${CXX} -o ${EXECUTABLE} ${OBJECTS}

main.o:main.cpp part1.cpp part2.cpp
    ${CXX} -c main.cpp

part1.o:part1.cpp part1.hpp
    ${CXX} -c part1.cpp

part2.o:part2.cpp part2.hpp
    ${CXX} -c part2.cpp

clean:
    rm -rf *.o && rm ${EXECUTABLE}
    @echo Done

help:
    @echo You can make:
    @echo all
    @echo clean
    @echo help
```

makefiles for parts

```
stud@stud-virtual-machine:~/maketest3.2$ ls
main.cpp Makefile Makefile~ part1.cpp part1.hpp part2.cpp part2.hpp
stud@stud-virtual-machine:~/maketest3.2$ make -f Makefile
g++ -o RUN main.cpp part1.cpp part1.hpp part2.cpp part2.hpp
stud@stud-virtual-machine:~/maketest3.2$ ls
main.cpp Makefile Makefile~ part1.cpp part1.hpp part2.cpp part2.hpp RUN
stud@stud-virtual-machine:~/maketest3.2$ ./RUN
This is part 1!
This is part 2!
stud@stud-virtual-machine:~/maketest3.2$ make help
You can make:
all
clean
help
stud@stud-virtual-machine:~/maketest3.2$ make clean
rm -rf *.o && rm RUN
Done
stud@stud-virtual-machine:~/maketest3.2$ ls
main.cpp Makefile Makefile~ part1.cpp part1.hpp part2.cpp part2.hpp
stud@stud-virtual-machine:~/maketest3.2$ make all
g++ -o RUN main.cpp part1.cpp part1.hpp part2.cpp part2.hpp
stud@stud-virtual-machine:~/maketest3.2$ ls
main.cpp Makefile Makefile~ part1.cpp part1.hpp part2.cpp part2.hpp RUN
stud@stud-virtual-machine:~/maketest3.2$
```

The terminal output for parts

The above is the terminal output from executing the makefile. All the commands gave the expected outcome.

Exercise 2.3.2 : Using pattern matching rules

The makefile created in the previous exercise is very explicit and rather large.

In this exercise the idea is to use the same but shrink it down and make it less error prone.

In make the solution is to use pattern-matching

which has special syntax involving the % character for representing wildcards.

In other words, one writes a general rule that applies to many situations alleviating the need to write a rule for each and every file.

In this version of the makefile two extra variables are needed:

- Source files
- Object files (acquired from the source file variable - how?)
- **Write an pattern matching rule that creates object files based on our cpp files.**

```
SOURCE=main.cpp part1.cpp part2.cpp
```

```
OBJECTS=${SOURCES:-cpp=.o}
```

```
EXECUTABLE=RUN
```

```
CXX=g++
```

```
all:${EXECUTABLE}
```

```
${EXECUTABLE}:${OBJECTS}
```

```
    ${CXX} -o ${EXECUTABLE} ${OBJECTS}
```

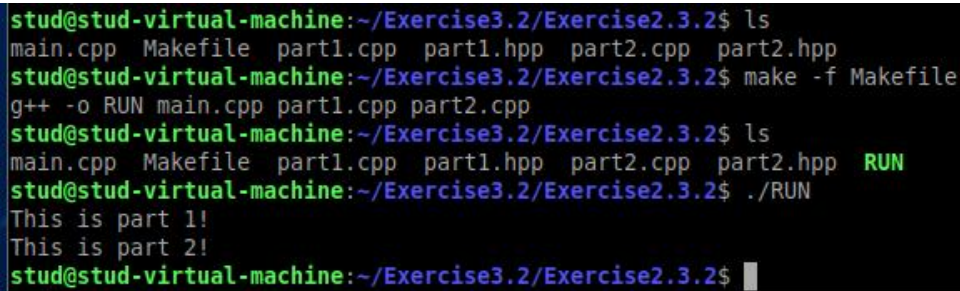
```
%.o:%.hpp
```

```
clean:
```

```
rm -rf *.o && rm ${EXECUTABLE}
@echo Done
```

```
help:
```

```
@echo You can make:
@echo all
@echo clean
@echo help
```



```
stud@stud-virtual-machine:~/Exercise3.2/Exercise2.3.2$ ls
main.cpp Makefile part1.cpp part1.hpp part2.cpp part2.hpp
stud@stud-virtual-machine:~/Exercise3.2/Exercise2.3.2$ make -f Makefile
g++ -o RUN main.cpp part1.cpp part2.cpp
stud@stud-virtual-machine:~/Exercise3.2/Exercise2.3.2$ ls
main.cpp Makefile part1.cpp part1.hpp part2.cpp part2.hpp RUN
stud@stud-virtual-machine:~/Exercise3.2/Exercise2.3.2$ ./RUN
This is part 1!
This is part 2!
stud@stud-virtual-machine:~/Exercise3.2/Exercise2.3.2$
```

- What makes this an improved solution as opposed the previous one?
 - When using the pattern matching rule, you'll be able to execute you code with a lot less work, by making the makefile create your headerfiles for you.

Exercise 2.4 : Problem...

The below makefile snippet compiles and produces a working executable. This is obviously assuming that the said files exist and are adequately sane. <- mood
In this particular scenario it is assumed that the following files exist:

- server.hpp & server.cpp
- data.hpp & data.cpp
- connection.hpp & connection.cpp

Listing 2.1: Simple makefile creating a simple program executable called prog:

```
1 EXE=prog
2 OBJECTS=server.o data.o connection.o
3
4 $(EXE): $(OBJECTS)
5     $(CXX) -o $@ $^
```

Questions to consider:

How are the source files compiled to object files, what happens?

- The Source files are compiled automaticly, because the compiler knows that it needs to make the needed object-files out of cpp-files.

When would you expect make to recompile our executable prog - be specific?

- Only when a change is made to a .cpp file : This gives the .cpp file a newer timestamp than the .o file, and hence re-builds.

Make fails using this particular makefile in that not all dependencies are handled by the chosen approach. Which ones are not ?

- .hpp files are not counted as sources by default, so a change in a .hpp file, will not force make to re-build.

Why is this dependency issue a serious problem?

- Because a change in a .hpp file can be just as important as a change in a .cpp file.

Exercise 2.5 : Solution

Analyze the listing:

Listing 2.2: Using finesse to ensure that dependencies are always met

```

1 SOURCES=main.cpp part1.cpp part2.cpp
2 OBJECTS=$(SOURCES:.cpp=.o)
3 DEPS=$(SOURCES:.cpp=.d)
4 EXE=prog
5 CXXFLAGS=-I.
6
7 $(EXE): $(DEPS) $(OBJECTS)    # << Check the $(DEPS) new dependency
8     $(CXX) $(CXXFLAGS) -o $@ $(OBJECTS)
9
10 # Rule that describes how a .d (dependency) file is created from a .cpp
    file
11 # Similar to the assignment that you just completed %.cpp -> %.o
12 %.d: %.cpp
13     $(CXX) -MT$(@:.d=.o) -MM $(CXXFLAGS) $^ > $@
14
15 -include $(DEPS)

```

Describe and verify what it does and how it alleviates our prior dependency problems!
In particular what does the command do?:

\$(CXX) -MT\$(@:.d=.o) -MM \$(CXXFLAGS) \$(SOURCES)

This code considers all sources, that the object-files are dependent on.

-MT\$(@:.d=.o): - MT is used to change the target. Here, it changes the target to .o-files instead of .d-files.

-MM: -MM changes the output to a rule, instead of a file.

All in all, the target will be a .o-file with a rule on how to make the .d-file.

Filer

Listing2.1.png	8,55 KB	2018-09-05	Brian Nymann
Listing2.2.png	38,8 KB	2018-09-05	Brian Nymann
Target_dep.png	51,2 KB	2018-09-05	Brian Nymann
Makefile_hello.png	39,1 KB	2018-09-05	Brian Nymann
Run_makefile_hello.png	40,5 KB	2018-09-05	Brian Nymann
Parts.png	26,1 KB	2018-09-05	Brian Nymann
Make_parts.png	68,4 KB	2018-09-05	Brian Nymann
makefile_parts.png	55 KB	2018-09-05	Brian Nymann
Make_parts2.png	35 KB	2018-09-05	Brian Nymann
makefile_parts2.png	45,4 KB	2018-09-05	Brian Nymann
Screenshot1.png	61 KB	2018-09-10	Brian Nymann