

Using Makefiles

When building large, maintainable C programs (with multiple entry points and shared code), the use of Makefiles is paramount.

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

These files are needed:

- *main.cpp*
- *hello.cpp*
- *factorial.cpp*
- *functions.h*

Create a new directory and place all the files in there.

Note: g++ is used for compiling.

The make utility

If you run

```
make
```

This program will look for a file named *makefile* in your directory, and then execute it.

If you have several makefiles, then you can execute them with the command:

```
make -f MyMakefile
```

There are several other switches to the *make* utility. For more info, *man make*.

Build Process

1. Compiler takes the source files and outputs object files
2. Linker takes the object files and creates an executable

Compiling by hand

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

```
g++ main.cpp hello.cpp factorial.cpp -o hello
```

The basic Makefile

The basic makefile is composed of:

```
target: dependencies
```

```
[tab] system command
```

This syntax applied to our example would look like:

```
all:
```

```
    g++ main.cpp hello.cpp factorial.cpp -o hello
```

To run this makefile on your files, type:

```
make -f Makefile-1
```

On this first example, our target is called *all*. This is the default target for makefiles. The *make* utility will execute this target if no other one is specified. We also see that there are no dependencies for target *all*, so *make* safely executes the system commands specified.

Finally, make compiles the program according to the command line we gave it.

Using dependencies

Sometimes is useful to use different targets. This is because if you modify a single file in your project, you don't have to recompile everything, only what you modified. Here is an example:

```
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
```

Now we see that the target *all* has only dependencies, but no system commands. In order for *make* to execute correctly, it has to meet all the dependencies of the called target (in this case *all*).

Each of the dependencies is searched through all the targets available and executed if found.

In this example we see a target called *clean*. It is useful to have such target if you want to have a fast way to get rid of all the object files and executable.

Using variables and comments

When writing Makefiles, variables can be used. It comes in handy in situations where you want to change the compiler, or the compiler options.

```
# I am a comment, and I want to say that the variable CC will be the compiler to use
```

```
CC=g++
```

```
# CFLAGS will be the options I'll pass to the compiler
```

```
CFLAGS=-c -Wall
```

```
all: hello
```

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

```
    $(CC) main.o factorial.o hello.o -o hello
```

```
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 *o hello
```

As you can see, variables can be very useful sometimes. To use them, just assign a value to a variable before you start to write your targets. After that, you can just use them with the dereference operator `$(VAR)`.

Reference:

<http://mrbook.org/blog/tutorials/make/>