

The consequence of modifications

Lecture 3

Automatic updating: make

Now programs do not mean a single file to us, but a set of related files.

By breaking a program into small parts, it is now possible to update the whole project by compiling only the affected parts of the program.

But it is usually very troublesome to figure out what files need recompilation. This trouble is best left to a good tool. We introduce one here: make.

References:

- Info pages of (**make**)

Once a program is broken up into many source files, we can **trim down the number of compilations** if only a few source files are modified.

Being able to do this is one of the primary reasons for separate compilation.

Example: suppose a program *main* is made up of *complex.cc*, *complex.h* and *main.cc*, with object files *main.o* and *complex.o*. *complex.h* is included by both *.cc* files.

- If you modified *complex.cc*, you need rebuilding *complex.o* and *main*.
- If you modified *main.cc*, you need rebuilding *main.o* and *main*.
- If you modified *complex.h*, you must rebuild everything.
This is header dependencies: modifying a header file is usually costly.

We won't want to do all these ourselves. We want automation. Otherwise, just figuring out what commands to use and which file to recompile will spend you more time than compiling!

PMOOP(0396A)

PMOOP(0396A)-3.1

Modified files

So we want automation. The process would be like:

- Modify your source files
- Type one command. The needed recompilations are found based on what files are modified. If everything goes well, we get a new program.

But wait... how the command we use know **which file is modified?**

No, comparing with the old file is not possible: the old file is probably gone.

The answer lies in a small piece of data that the OS keeps track of: **file modification date** (the latest *time* when the file content changes).

If you think that other times are interesting as well, the OS also keeps the latest time when file mode changes and when file is accessed.

If the **date of a target file** is **earlier** than its sources ("**pre-requisites**"), the pre-requisites are considered to be modified.

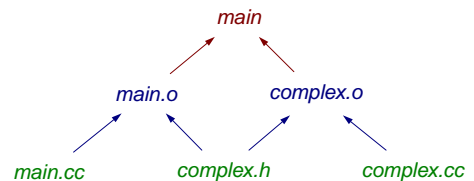
There is one rare race condition here, can you find it?

PMOOP(0396A)-3.2

PMOOP(0396A)-3.3

File dependencies

Logically, the files are related by a **dependency graph**:



For example, *complex.cc* and *complex.h* are pre-requisites of *complex.o*, meaning that if we change *complex.cc* or *complex.h*, we need to rebuild *complex.o*.

We can also say that *complex.o depends on complex.cc and complex.h*.

This will subsequently cause *main* to be rebuilt as well.

Note that we need to check *main.o* and *complex.o* before *main*.

PMOOP(0396A)-3.2

PMOOP(0396A)-3.3

The utility "make"

That's the basics. We now need a utility that read our specification of file dependencies, check the file dates, and update the files in the right order.

The utility is **make**, and the dependency specification is a **Makefile**. A very simple Makefile:

```
# Sample Makefile
main: main.o complex.o
    g++ -Wall -o main main.o complex.o

main.o: main.cc complex.h
    g++ -Wall -c main.cc

complex.o: complex.cc complex.h
    g++ -Wall -c complex.cc
```

There are 3 groups of lines here, each is called a **rule**.

PMOOP(0396A)-3.4

PMOOP(0396A)-3.5

What it really mean?

Let's focus on one rule and see what it means:

```
main: main.o complex.o
    g++ -Wall -o main main.o complex.o
```

The rule has two lines.

- The first line contains a **target**, followed by a colon and then a **list of pre-requisites**. It encodes the "edges" of the dependency graph.
The target part can be a list as well, specifying common dependencies. Such usage is less common, but we will see one later.
- The second line contains a **command**. To tell **make** that it is not the start of another rule, the first character **must** be a **tab** (not 8 spaces).

It means: *To update main, first update main.o and complex.o. Then, if either has file time later than main, run the command g++ -Wall -o main main.o complex.o.*

PMOOP(0396A)-3.4

PMOOP(0396A)-3.5

Some pitfalls

Pitfall: We run the Makefile

The Makefile is **not a program** and cannot be executed. It just **specifies dependencies** and **how to rebuild targets** when out-dated.

Pitfall: We must first specify how to rebuild the prerequisites

Rules of Makefile can be **in any order**, and it doesn't matter at all. With one exception to be explained soon.

Pitfall: make can only be used for programming

While make is most frequently used for programs, it is useful whenever the building process can be **speed-up by running fewer commands**.

E.g., this lecture notes is created by make.

Pitfall: Each object file depends on all headers.

Each object file depends on those files used to generate it. In other words, the source file and those that are *#include*'d into the source file.

But also those included indirectly: *#include* in an include file also counts.

Pitfall: Makefile is created after most code is there.

Makefile assists your development, so it should be created ASAP.

PMOOP(0396A)-3.6

How to use the Makefile

Once we have Makefile, we can update our programs by typing

```
make target
```

where *target* is the **ultimate file that you want to update**, e.g., *main*. If you don't specify it (typing just *make*), the target of the first rule is used.

This is the exception that order does matter: first rule is the default.

Then *make* will perform the followings:

- To update *main*, *main.o* and *complex.o* are first **recursively updated**.
- To update *main.o*, *main.cc* and *complex.h* are checked. If either is newer than *main.o*, or if *main.o* is missing, update.
- Similar operations are done for *complex.o*.
- If either *main.o* or *complex.o* is newer than *main*, or if *main* is missing, update *main*.

PMOOP(0396A)-3.7

Implicit rules (pattern rules)

Usually the command part can be automatically generated. E.g.: for any *input*, to build *input.o* from *input.cc*, we use `g++ -Wall -c input.cc`.

In such cases we can use an **implicit rule** to specify all commands for them. Such rules are of the following form:

```
%.o: %.cc
    g++ -Wall -c $<
```

There are a few things to notice:

- The first line of the rule is very similar to an explicit rule, except that part of it contains **the character % specifying a wildcard**.
- The command part does not use the % character. Instead, it has a **special variable "\$<"** that represents the left-most pre-requisite.
- Other useful character: **\$^** represents all pre-requisites, **\$@** represents the target.

PMOOP(0396A)-3.8

Makefile using implicit rule

With implicit rule, the Makefile looks like this. It isn't a big improvement, but think about what you get when you have 20 source files!

In this course, always think large.

```
# Sample Makefile
main: main.o complex.o
    g++ -Wall -o main main.o complex.o

main.o: main.cc complex.h
complex.o: complex.cc complex.h

%.o: %.cc
    g++ -Wall -c $<
```

But it is possible to make it even simpler...

PMOOP(0396A)-3.9

Built-in rules

There are some rules already defined when make start executing. E.g.

```
%.o: %.cc
    $(CXX) $(CXXFLAGS) $(CPPFLAGS) -c -o $@ $<
```

where **\$(CXX)** etc are **make variables** with some sensible defaults.

For other pre-defined rules, see the info pages under Catalogue of Implicit Rules.

So a reasonable Makefile looks like this:

```
main: main.o complex.o
    $(CXX) -o $@ $^
main.o: main.cc complex.h
complex.o: complex.cc complex.h
```

With the default values, the commands that get executed are

```
g++ -c -o main.o main.cc
g++ -c -o complex.o complex.cc
g++ -o main main.o complex.o
```

PMOOP(0396A)-3.10

Magic?

One consequence of pre-defined implicit rules is that **you can use make without a Makefile**.

Internally, make has a rule like this:

```
%.o: %.cc
    $(CXX) $(CXXFLAGS) $(CPPFLAGS) $(LDFLAGS) -o $@ $<
```

If you have *myprog.cc*, and want to compile it, you just need:

```
make myprog
```

without even a Makefile! The command used:

```
g++ -o myprog myprog.cc
```

Note that searching for a good source file is automatic.

Question: why we need the command for the main program in our Makefile? Try to delete it and see what happen.

PMOOP(0396A)-3.11

Non-file targets

One can also use `make` for **convenience commands**. E.g., it is common to add a rule like this:

```
clean:
    rm -f comp comp.o complex.o
```

So if you type `make clean`, the generated files are all removed. Do this when you want to free up disk space.

This normally works, since the file named “clean” does not exist, and so `make` tries to create it by running the command. But it’s a bit strange: **why asking `make` to check for the known non-existent file?**

And what if a file called `clean` is really there?

The special `.PHONY` rule tells `make` to skip the file date checking:

```
.PHONY: clean
Phony means something that is not real.
```

PMOOP(0396A)-3.12

Using “variables”

Now, suppose you want to optimize your program, by **adding the `-O2` flag** to the compilation **command**. How to do it?

The problem: we don’t have a place to write our command!

Still remember that there are **make variables** like `$(CXXFLAGS)` in the command part of the pre-defined rule? They are provided for this.

But “variable” is not in the sense of a programming language: Makefile is not a program. Perhaps “macro” is a better word.

For example, if we want to add “`-Wall -O2`” to the compile command, do this:

```
CXXFLAGS = -Wall -O2
```

This is also modified by the **environment variable** of the same name. E.g., if suddenly you want one compilation with “`-Wall -g`”, you can type

```
CXXFLAGS="-Wall -g" make
i.e., without modifying the Makefile.
```

PMOOP(0396A)-3.13

What Makefile we got now

Now we also want to create a variable for our own use, since we don’t want the list of object files to appear twice.

(Once in pre-requisite, once in clean rule.)

```
OBJS          = main.o complex.o
CXXFLAGS      = -Wall -O2

.PHONY: clean
main: $(OBJS)
    $(CXX) -o $@ $^

clean:
    rm -f main $(OBJS)
main.o: main.cc complex.h
complex.o: complex.cc complex.h
```

So it starts to look like a manageable and **scalable** system: everytime we add a new file, we just add a new object into the `OBJS` line.

PMOOP(0396A)-3.14

Generating dependencies automatically

But there is one problem: we have to **keep writing dependencies**.

It changes whenever we add a `#include` somewhere.

We would like to **automate** the **update of dependencies** as well.

There is one feature of `g++` which is very useful when used with `make`: the `-MM` option. Let’s see what it does:

```
> g++ -MM complex.cc
complex.o: complex.cc complex.h
```

In other words, the **compiler** reads through the source file, interprets it, and **outputs the dependency** (instead of a compiled object).

Now it seems like that we don’t have to learn what is dependencies after all.

PMOOP(0396A)-3.15

Global dependencies

How to use it in our Makefile? Of course, we can just **delete all the dependency lines** (except that for the program file) and type

```
g++ -MM *.cc >> Makefile
```

everytime we want to update the dependencies, but this is **troublesome**.

We want the “deleting the dependency lines” and “typing command” parts automated, but this it is difficult to automatically edit a file reliably.

An easier solution is to have a **separate file** for all such dependencies (let’s say, `.depend`).

Everytime we want to regenerate the dependencies, we simply **overwrite** the old `.depend`. Of course, done by another phony rule.

The Makefile should use an `include` directive, telling `make` to read it.

```
-include .depend
(-include ignores non-existent file. If you want error instead, delete the hyphen.)
```

PMOOP(0396A)-3.16

Makefile at this time

After a few more modifications:

```
PROG          = main
OBJS          = main.o complex.o
SRCS          = main.cc complex.cc
CXXFLAGS      = -Wall -O2

$(PROG): $(OBJS)
    $(CXX) -o $@ $^

.PHONY: clean dep
clean:
    rm -f $(PROG) $(OBJS)

dep:
    g++ -MM $(SRCS) > .depend
-include .depend
```

Seems complicated, but think about the improvement we achieved.

During development, all modifications needed are in first 3 lines—and are trivial!

PMOOP(0396A)-3.17

What our Makefile does now

Once we got the Makefile ready, the following things can be done when we write and modify our program.

- make dep: **re-create the dependency file** .depend; should be done first, and every time when any `#include` is modified.
- make: **re-create the program file**; should be done after modifying any source or header.
But after “make dep” is executed, in case that is needed.
- make clean: clean-up the directory.
One might want to clean-up the dependency file as well.
- **make a sub-target**, e.g., make complex.o: used very rarely, just check whether compilation would succeed.
- **Edit the Makefile**, when a new object file is created. We need to modify both the SRCS and OBJS lines.

PMOOP(0396A)-3.18

Substitutions in make

You might think that it is a good idea to replace the list of source files by *.cc, but resist that temptation.

Why? Sometimes you will create a testing .cc file in your directory, and don't want “make” to process it.

But on the other hand, if the SRCS and the OBJS are that similar, **it is a good idea to generate one from another**.

How? In the Makefile you can write the following:

```
OBJS          = main.o complex.o
SRCS          = $(OBJS:.o=.cc)
```

Here `$(OBJS:.o=.cc)` recall the value of OBJS, and replace all the trailing .o by .cc. Of course, this is exactly what we want.

Recall that in our model, we have one .o file per .cc file, and they are supposed to have the same name other than the extension.

PMOOP(0396A)-3.19

Advanced usage: no more “make depend”

- For most projects, such a Makefile is all that is needed.
For larger projects that this doesn't suffice, read the info pages!
- But to **illustrate the power of make**, let's do one step further: **we want no “make dep”**.
- In other words, the dependencies should be re-generated everytime **automatically** when it is needed, and only when needed.
- To make it possible to re-create only some (but not all) dependencies, we will have **one dependency file per source file**, let's say “file.d”.
- Clearly, if a **source file is modified**, the .d file needs updating.
Expect a rule that the .d file depends on the .cc file.
- Less clearly, if a **related header file is modified**, the .d file also needs to be updated.
This is due to the possibility of nested inclusion.

PMOOP(0396A)-3.20

Dependency files

So what should be the dependency for a .d file, say main.d?

- Possible line:

```
main.d: main.cc complex.h
```
- It might seems that this is a *chicken and egg problem*: to automate dependency generation, we need **new dependencies**.
- But it is not: note that the dependency is just the same as main.o, so we can re-use the output of `g++ -MM main.cc`.
- So now the .d file contains the dependencies for the .o file **and** the .d file, in the following form:

```
main.d \  
main.o: main.cc complex.h
```

Backslash is line continuation, “make” ignore it together with the next newline.
- So it really just requires two simple commands.

PMOOP(0396A)-3.21

Our final Makefile

Our final *Makefile* looks like the following:

```
PROG          = main
OBJS          = main.o complex.o
DEPS         = $(OBJS:.o=.d)
CXXFLAGS     = -Wall -O2

$(PROG): $(OBJS)
    $(CXX) -o $@ $^

%.d: %.cc
    echo $@ \> $@
    g++ -MM $< >> $@

.PHONY: clean
clean:
    rm -f $(PROG) $(OBJS) $(DEPS)
    include $(DEPS)
```

Double backslash in the %.d rule is needed by the shell program.

PMOOP(0396A)-3.22

Advices

- Whenever you start a project with multiple source files, you will need some system to manage them.
- Files in a project are typically related by dependencies: when a header file is modified, more than one file needs to be updated.
- Different programming environments use different tools to help you run the needed commands, but the basics are just the same.
- The setup should be done once your project starts.
- Make sure you understand the following in make:

Explicit and implicit rules, built-in rules, dependencies and pre-requisites, make variables, g++ -MM flags, substitutions.

Exercise: write a *Makefile* for the files you've made in the exercise of the last lecture. (Most of it can be very similar to our Makefile here.)

PMOOP(0396A)-3.23