

**The consequence of modifications**

**Lecture 9**

**Automatic updating: make**

We programs do not mean a single file to us, but a set of related files, some made by library writers, some made by library users.

By breaking a command into small chunks, 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 troublesome task is best left to a good tool.

References:

- Info pages of **(make)**

Once a program is broken up into many source files, different kind of modifications requires different actions.

In our last example: our program is made up of *complex.cc*, *complex.h* and *comp.cc*.

- If you modified *complex.cc*, you must build *complex.o* and *comp* again.
- If you modified *comp.cc*, you must build *comp.o* and *comp* again.
- If you modified *complex.h*, you must build *complex.o*, *comp.o* and *comp* again.

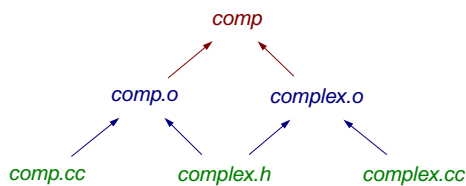
It is very easy to forget some of the needed compilations. It would be nice if we have a tool to do this automatically, so that only the necessary recompilations are made.

PMOOP(0396A)

PMOOP(0396A)-9.1

**File dependencies**

Logically, the files are related by a dependency graph:



For example, *complex.o* depends on (or results from) *complex.cc* and *complex.h*, so we change *complex.cc*, we need to rebuild *complex.o*.

If we modify *complex.cc*, we also need rebuilding *comp*, but we don't need a link from *complex.cc* to *comp*: changing *complex.cc* would cause *complex.o* to be rebuilt, which causes *comp* to be rebuilt.

PMOOP(0396A)-9.2

**The utility "make"**

The *make* utility reads a *Makefile* (specifying dependencies), detect changes to files by checking modification dates of files, and runs the specified commands to make sure the *target* file is up-to-date.

An example Makefile

```

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

**A rule specifies the dependencies and specifies commands for updating**

**Tabs distinguish between new rule and commands.**

PMOOP(0396A)-9.3

**A run of make**

When we run "make" in a directory containing a *Makefile*, the following happens:

The left hand side of the first rule, *comp*, is used as the ultimate target.

To update *comp*, *comp.o* and *complex.o* are checked.

To update *comp.o*, *comp.cc* and *complex.h* are checked.

If either is newer than *comp.o*, or if *comp.o* is missing:  
`g++ -Wall -c comp.cc` is executed.

To update *complex.o*, *complex.cc* and *complex.h* are checked.

If either is newer than *complex.o*, or if *complex.o* is missing:  
`g++ -Wall -c complex.cc` is executed.

If either *comp.o* or *complex.o* is newer than *comp*, or if *comp* is missing:  
`g++ -Wall -o comp comp.o complex.o` is executed.

Note that this always gives a completely updated *comp* program.

PMOOP(0396A)-9.4

**Pattern rules**

Usually the command part can be automatically generated. For example, to build most *.o* files from *.cc* files, we use `g++ -Wall -c input.cc`.

In such case we can use a pattern rule to specify all the rules from *.cc* to *.o* files:

```

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

**Implicit rule specifies commands**

**\$< means the first source file**

**Rules only specify dependencies**

If *make* cannot find a command in the rule, it will try to create a rule by using the pattern rule, replacing *\$<* by the input filename, and *#{@}* by the output filename.

PMOOP(0396A)-9.5

### Making non-default target

You can invoke “make” like “make target” in order to specify exactly the target you want to make, instead of the left-hand side of the first (non-pattern) rule.

For example, `make complex.o` will only build `complex.o`, without touching `comp.o` or `comp`.

This is usually used for some convenience target. For example, if we have the following in the `Makefile`, we can run `make clean` and remove all the generated files:

```
.PHONY: clean

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

The special `.PHONY` rule makes sure that even if there is a file called `clean`, the command will be executed. Simply list all such targets in the right hand side of the `.PHONY` rule.

### Generating dependencies automatically

One problem when using `make` is that people keeps forgetting to put new needed headers into the `Makefile`. Of course, if the `Makefile` is incorrect, we cannot hope that the target is updated correctly.

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

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

In other words, it outputs the needed dependency line for the C++ source file.

That is, *we do not really need to write dependency lines for C++ source files!* Making this automatic is a bit challenging, though. We want to *update the `Makefile` itself when the C++ files are updated!*

Instead of this strange thing, we will *include* other “dependency files” from the `Makefile`.

### Dependency files

We will ask `make` to generate dependency lines for, e.g., `comp.cc` in a “dependency file” like `comp.d`. This is easy with a simple pattern rule:

```
%.d: %.cc
    g++ -Wall -MM $< > $@
```

Then we ask `make` to include these files with an “include directive”:

```
include complex.d comp.d
```

The `make` program will automatically update `complex.d` and `comp.d`, and possibly reread the `Makefile`, before working on the target:

```
>make
Makefile:14: complex.d: No such file or directory
Makefile:14: comp.d: No such file or directory
g++ -Wall -MM comp.cc > comp.d
g++ -Wall -MM complex.cc > complex.d
g++ -Wall -c comp.cc
g++ -Wall -c complex.cc
g++ -Wall -o comp complex.o comp.o
```

*Makefile is reread after this to get new dependencies.*

### Variables in make

Now our `Makefile` contains quite a few occurrences of the list of `.o` files. All these need to be updated whenever we add a file to our project: not very good idea. Make have variables. If we have a line like `VAR = val`, every occurrence of `$(VAR)` will expand to `val`. So we can do the following:

```
OBJS = complex.o comp.o
DEPS = complex.d comp.d
comp: $(OBJS)
    g++ -Wall -o comp $(OBJS)
clean:
    rm -f comp $(OBJS) $(DEPS)
include $(DEPS)
```

Even the `DEPS` variable can be automatically generated:

```
DEPS = $(OBJS:.o=.d)
```

This means that `DEPS` should have the value of the variable `OBJS`, replacing all trailing `.o` by `.d`.

### Our final Makefile

Our final `Makefile` looks like the follows:

```
OBJS = complex.o comp.o
DEPS = $(OBJS:.o=.d)

.PHONY: clean

%.d: %.cc
    g++ -Wall -MM $< > $@
%.o: %.cc
    g++ -Wall -c $<

comp: $(OBJS)
    g++ -Wall -o comp $(OBJS)

clean:
    rm -f comp $(OBJS) $(DEPS)

include $(DEPS)
```

### Summary

The `make` utility comes in handy for separate compilation.

- Files in a project are typically related by dependencies, specifying what need to be updated when a file is modified.
- The `make` utility helps you by reading the description of dependencies in terms of rules, and execute associated commands to update a project.
- Pattern rules associate a default command to a pattern of dependencies, and phony rules are ones that does not create a file.
- The `g++` compiler has a flag to help you creating dependencies automatically. Variables allow easy management of `Makefile`'s.

**Exercise:** write a `Makefile` for the files you’ve made in the exercise of the last lecture. Look at the source of this lecture to see how different I do it, and argue about which way is better.

## Lecture 10

### Versions Control

Software development is a long and complicated process. The process is more complicated when there are multiple developers.

Many tasks require knowledge of the software history. CVS is a system helping us to keep track of it.

#### References:

- Info pages of (**cvs**). In the Overview node you will find a very short tutorial.

PMOOP(0396A)

PMOOP(0396A)-10.1

#### An alternative solution: overview of version control

- We keep the latest version of all source files in a “repository”.
- We also keep the modifications (diffs) made in each version of each file in the repository. (More exactly, we keep the reverse diffs, the modification needed to get back the old version.)
- Every time a new internal version is made, we update the repository. We call the operation “**checking in**” the changes.
- At any time we can get any version from the repository, by getting the latest version and using the diffs to regenerate the version. We call this “**checking out**” the source files.
- *Each file is handled independently.* Changing one file requires us to update the modification information of one file only.
- The independent version numbers of input files are related by giving a symbolic name, or **tag**, to a version of each source file.

PMOOP(0396A)-10.2

PMOOP(0396A)-10.3

#### Creating a repository

To use CVS, the first thing to do is to create a repository, i.e., a directory for CVS to store all the files and diffs.

- An environmental variable **CVSROOT** control the directory which CVS uses as a repository. You can use any name, but it is better to avoid the names **CVS** and **CVSROOT**, since they have special meaning in CVS.
- To set it, run the bash command
 

```
export CVSROOT=DIR
```

 where **DIR** is the directory to use for the repository. You need to set **CVSROOT** everytime you login or make a new terminal window.
- Then you can run
 

```
cvs init
```

 to initialize the repository. You can see that the repository contains only one directory, **CVSROOT**, which is the configuration directory of CVS.

PMOOP(0396A)-10.4

PMOOP(0396A)-10.5

#### Capability of CVS

All these things will be very tedious if we have to do it manually. So instead we use a program to do them: **CVS** (Concurrent Versions System). Here are its capabilities:

- Organize the repository, allow simple commands to operate on them.
- Allow modifications to old version, for example to fix a bug of the old version of the software without touching the latest version.
- Find the difference between versions of a source file.
- Patch a new version with the difference found.
- Find out the version in which each line of a source file first appears.
- Allow multiple people to work together on the same project, and notify a programmer if his changes conflict with that of another.

#### Adding an existing project to the repository

Once you have a repository, you store projects in it.

- First, change to the directory storing your project. If you want to create a completely new project, just change to a new empty directory.
- Now let's store the project in a directory **myproj** within the repository.
 

```
cvs import myproj myproj start
```
- CVS will ask for comments with **vi** (or the editor you set with environment variable **EDITOR**), before adding the directory to the repository.
- At this time, the directory is **not** managed by CVS. So change to the parent directory, delete the directory and immediately **check-out** a new directory that is managed by CVS.
 

```
cvs co myproj
```
- A directory **CVS** records the status of the newly created directory.

## Basic operation on repository

In the working directory, we can do things like:

Operation	Command
Check in changes	<b>cv</b> s ci
Add some files <i>file1</i> , <i>file2</i> , ... to repository	<b>cv</b> s add <i>file1</i> , <i>file2</i> , ...
Remove some files <i>file1</i> , ... from repository	<b>cv</b> s remove <i>file1</i> , ...
Find modified files since latest version	<b>cv</b> s update
Look at version logs	<b>cv</b> s log [ <i>file</i> ]
Look at an annotated source file <i>file</i>	<b>cv</b> s annotate <i>file</i>

If we want to delete the directory, it is best to make sure that no file is modified in the working directory. We do both operations by changing to the parent directory and use

**cv**s release -d myproj

PMOOP(0396A)-10.6

## Files to ignore

- We usually have some files in the working directory that we don't want to keep in repository. For example, backup files and compiled files normally shouldn't be in the repository.
- They pose no problem, except that **cv**s update gives an warning.
- To avoid this, list the patterns of such files in a file .cvsignore in the root of the working directory, and add that file to the repository. For example, .cvsignore might contain this:

```
*~
*.o
a.out
```

PMOOP(0396A)-10.7

## The game of versions

- Each time we check in a source file, a new (internal) version is created, and a new version number is automatically created.
- Then how about the numbers that we assign to some version of the entire software which is released?
- CVS uses a very general scheme for these: tags. A symbolic tag (containing alpha-numeric characters, hyphens and underscores) can be assigned to a version of each source file.
- If we want to release the latest repository version as release 2.0, we might assign a tag rel-2-0 to all source files in the current directory:

**cv**s tag rel-2-0 .

- To check out a particular release, you can use -r. E.g.,

**cv**s co -r rel-2-0 myproj

PMOOP(0396A)-10.8

## Role of Symbolic version

Symbolic versions (tags) are used to give an idea about which versions of source files form a consistent release. A picture illustrate this.

File A	File B	File C	Tag
1.1	1.1		
1.2	1.2	1.1	
	1.3		
1.3	1.4		
(1.3)	(1.4)	(1.1)	rel-0-9
1.4			
1.5		1.2	
(1.5)	(1.4)	(1.2)	rel-1-0

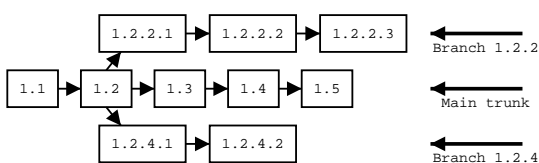
We use a symbol rel-0-9 instead of three numbers 1.3, 1.4 and 1.1.

PMOOP(0396A)-10.9

## Branching

Sometimes, after a release is made, and after new code has been checked in, the old release need modifications, e.g., to fix a serious bug.

History is then non-linear. CVS treats development histories as trees.



We call the main development line the trunk, and each other development line a branch. E.g., versions from 1.2.2.1 to 1.2.2.3 forms branch 1.2.2.

However, we seldom use the actual branch numbers. Instead, We usually use a tag for each branch, to manage the branches of all source files.

PMOOP(0396A)-10.10

## Using branches

To create a branch called rel-1-0-patches at rel-1-0, we do the following:

**cv**s tag -r rel-1-0 -b rel-1-0-patches

Or, if we want have already checked out rel-1-0, just do this:

**cv**s tag -b rel-1-0-patches

Then we can switch to the newly created branch:

**cv**s update -r rel-1-0-patches

At this point, any check-in will check into the branch rel-1-0-patches, not the main trunk. This is revealed by the status command:

**cv**s status

To switch back to the main trunk, do the following:

**cv**s update -A

PMOOP(0396A)-10.11

### Keywords

At some times we want the version information of files, e.g., the version number, to appear directly in the file.

CVS has a keyword substitution mechanism that automatically insert such information into the source files.

For example, we can add the following comment to the C++ source:

```
// $Id$
```

When the file is checked out, we get something like this:

```
// $Id: hello.cc,v 1.5 2001/2/19 14:57:32 kkto Exp $
```

This gives easy access to the version number of the file. We can even embed the file into the compiled executable: read the manual.

PMOOP(0396A)-10.12

### Multiple developers

What to do if there are multiple people working on the same code base?

- At the same time, there can be more than one person who has checked out the same directory of a repository.
- CVS uses a scheme called “unreserved check-out”, meaning that each of the developers can modify any file without notifying others about it.
- If two people are editing the same file, one will check-in the file too late. CVS will try its best to merge the changes.
- If that fails (we say, we have a conflicting change), the one who check-in late will get an error, and thus has to fix the problem before proceeding for check-in again.

In practice, this scheme works very well only if each developer is assigned a distinct task to do, so that changes are typically not in the same places. I.e., we need management even if we use CVS.

PMOOP(0396A)-10.13

### Summary and Exercise

- We usually need to keep history of source files. CVS allows this to be done conveniently and economically.
- A repository stores the latest versions of the source file, together with reverse diffs to generate previous versions.
- A Tag keeps the version numbers for a consistent set of files, and a branch allow previous version to be further developed.
- Multiple developers can check-out many directories, and CVS will try to merge changes if they modified the same file.

### Exercise

Read the info page of cvs to find out how to answer each question in page 9.1.

PMOOP(0396A)-10.14