Sviluppo software in gruppi di lavoro complessi[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

Anno accademico 2024/25, II semestre

[1] © ① ⓪ 2025 M. Monga. Creative Commons Attribuzione — Condividi allo stesso modo 4.0 Internazionale.
http://creativecommons.org/licenses/by-sa/4.0/deed.it

Lezione XII: Sistemi di *build automation*

Costruire (assemblare) un prodotto software fatto di molti componenti è tutt'altro che banale:

- dipendenze da componenti che non controlliamo (*dependency hell*)
- dipendenze fra componenti che stiamo sviluppando

Make

Stuart Feldman, 1977 at Bell Labs.
Permette di specificare dipendenze fra processi di generazione.
Dipendenze: se cambia (secondo la data dell'ultima modifica) un prerequisito, allora il processo di generazione deve essere ripetuto.

```
helloworld.o: helloworld.c
	cc -c -o helloworld.o helloworld.c

helloworld: helloworld.o
	cc -o $@ $<

.PHONY: clean
clean:
	rm helloworld.o helloworld
```

Make: come funziona

- Le dipendenze definiscono un grafo aciclico che ammette un unico *ordinamento topologico* (in quanto si passa una sola volta da ogni *target*)
- I processi di generazione (*recipes*) sono eseguiti seguendo l'ordinamento topologico
- Nei `make` moderni è possibile eseguire processi di generazione indipendenti in parallelo (`make -j`)

Parte del materiale che segue è preso da: http://www.lrde.epita.fr/~adl/autotools.html

Standard Makefile Targets

`make all`	Build programs, libraries, documentation, etc. (Same as `make`.)
`make install`	Install what needs to be installed.
`make install-strip`	Same as `make install`, then strip debugging symbols.
`make uninstall`	The opposite of `make install`.
`make clean`	Erase what has been built (the opposite of `make all`).
`make distclean`	Additionally erase anything `./configure` created.
`make check`	Run the test suite, if any.
`make installcheck`	Check the installed programs or libraries, if supported.
`make dist`	Create `PACKAGE-VERSION.tar.gz`.

Standard File System Hierarchy

Directory variable	Default value
prefix	/usr/local
exec-prefix	prefix
bindir	exec-prefix/bin
libdir	exec-prefix/lib
...	
includedir	prefix/include
datarootdir	prefix/share
datadir	datarootdir
mandir	datarootdir/man
infodir	datarootdir/info
...	

Portabilità della costruzione

Il modello di `make` assume un ambiente di *build* fisso.

- L'ipotesi è irrealistica perfino nel mondo dello sviluppo anni '70 (C/UNIX)
- Compilatori, librerie cambiano molto anche nell'ambito degli standard

Non-Portability in C

Funzioni di libreria che:

- non esistono ovunque (es. `strtod()`)
- hanno nomi diversi (es. `strchr()` vs. `index()`)
- hanno prototipi differenti (es. `int setpgrp(void);` vs. `int setpgrp(int, int);`)
- hanno comportamenti diversi (e.g., `malloc(0);`)
- richiedono diverse dipendenze transitive (`pow() in libm.so or in libc.so?`)
- richiedono diverso trattamento (`string.h` vs. `strings.h` vs. `memory.h`)

Le soluzioni in C

- `#if/#else`
- substitution macros
- substitution functions

Code Cluttered with #if/#else

```
#if !defined(CODE_EXECUTABLE)
  static long pagesize = 0;
#if defined(EXECUTABLE_VIA_MMAP_DEVZERO)
  static int zero_fd;
#endif
  if (!pagesize) {
#if defined(HAVE_MACH_VM)
      pagesize = vm_page_size;
#else
      pagesize = getpagesize();
#endif
#if defined(EXECUTABLE_VIA_MMAP_DEVZERO)
      zero_fd = open("/dev/zero", O_RDONLY,0644);
      if (zero_fd < 0) {
        fprintf(stderr, "trampoline: Cannot open /dev/zero!\n");
        abort();
      }
#endif
  }
#endif
```

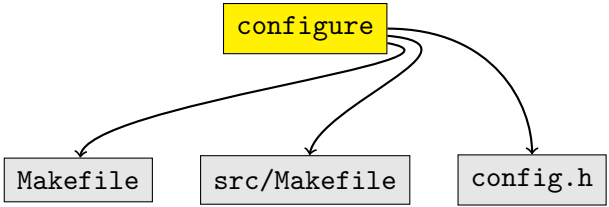
Caos. . .

A physicist, an engineer, and a computer scientist were discussing the nature of God. "Surely a Physicist," said the physicist, "because early in the Creation, God made Light; and you know, Maxwell's equations, the dual nature of electromagnetic waves, the relativistic consequences..." "An Engineer!," said the engineer, "because before making Light, God split the Chaos into Land and Water; it takes a hell of an engineer to handle that big amount of mud, and orderly separation of solids from liquids..." The computer scientist shouted: "And the Chaos, where do you think it was coming from, hmm?"

–Anonymous

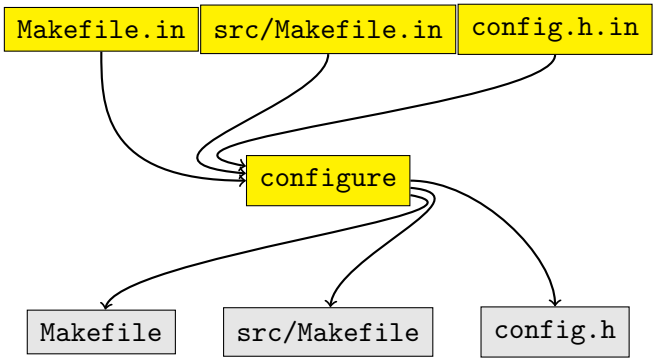
dal manuale di Autoconf

configure



- `configure` verifica le caratteristiche dell'ambiente di costruzione.
- genera un `config.h` con le `#define` giuste
- e un `Makefile`

configure process



`*.in` sono configuration templates da cui `configure` genera lo script di verifica dell'ambiente.

Ant

- Build automation: dipendenze + processi di generazione + riconfigurazione all'ambiente di *build*
- Java e XML
- Plugin in Java

Ant

```xml
<?xml version="1.0"?>
<project name="Hello" default="compile">
    <target name="clean" description="remove intermediate files">
        <delete dir="classes"/>
    </target>
    <target name="clobber" depends="clean" description="remove all artifact files">
        <delete file="hello.jar"/>
    </target>
    <target name="compile" description="compile the Java source code to class files">
        <mkdir dir="classes"/>
        <javac srcdir="." destdir="classes"/>
    </target>
    <target name="jar" depends="compile" description="create a Jar file for the application">
        <jar destfile="hello.jar">
            <fileset dir="classes" includes="**/*.class"/>
            <manifest>
                <attribute name="Main-Class" value="HelloProgram"/>
            </manifest>
        </jar>
    </target>
</project>
```

Prospettiva

Il supporto dei *tool* al processo di sviluppo:

Configuration management *artifact*, configurazioni, storia delle configurazioni

Build automation dipendenze fra *artifact*

- make: L'ambiente di sviluppo è implicito: dipendenze solo fra gli *artifact* sotto controllo
- ant/maven: grazie a cataloghi centralizzati, dipendenze su tutti componenti, l'ambiente di sviluppo è ancora implicito (ma *embedded*)
- gradle: anche l'ambiente di sviluppo è descritto nella "build automation", almeno in parte

Gradle

- Il processo di *build* è descritto con un linguaggio *general purpose* (e librerie di "*domain specific*", si tende comunque a considerarlo un *Domain Specific Language (DSL)*): Groovy, Kotlin
- *build-by-convention* (eventualmente configurabile)
- Supporta cataloghi di componenti
- Supporto per il test
- Reportistica

Gradle

```
// This is Groovy (build.gradle), Kotlin (build.gradle.kts) is also possible
plugins {
    id 'java'
    id 'application'
}

application {
    mainClass.set('it.unimi.di.svigruppo.Main')
}


jar {
    manifest {
        attributes 'Main-Class': application.mainClass
    }
}

version '1.2.3'

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.1'
}
```

Gradle “DSL”

```
task hello {

    group 'svigruppo'
    description 'Saluta lo sviluppatore'

    doLast {
        println 'Hello user!'
    }
}

task anotherHello {
    doFirst {
        println 'Salutoni!'
    }
}

anotherHello.dependsOn hello
```

Gradle “DSL” (cont.)

```
task copia(type: Copy) {
    from 'source'
    into 'destination'
}

task ciao(type: Exec) {
    workingDir '.'
    commandLine '/usr/bin/echo', 'ciao mamma'
}
```