Build Automation:
Introduzione a Make, Autotools e CMake

Luca Ceresoli
luca@lucaceresoli.net
http://lucaceresoli.net

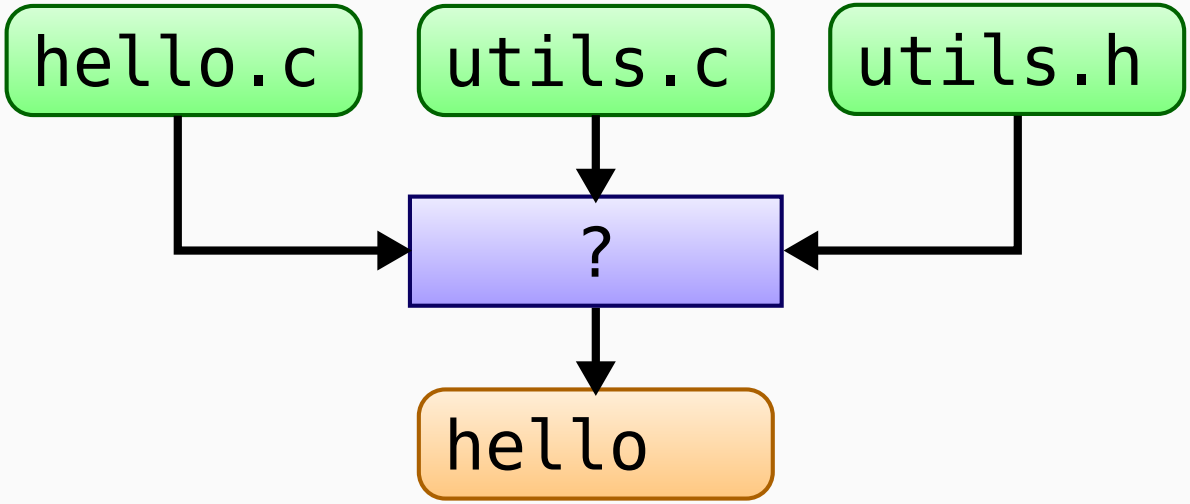
Linux Day 2017

Agenda

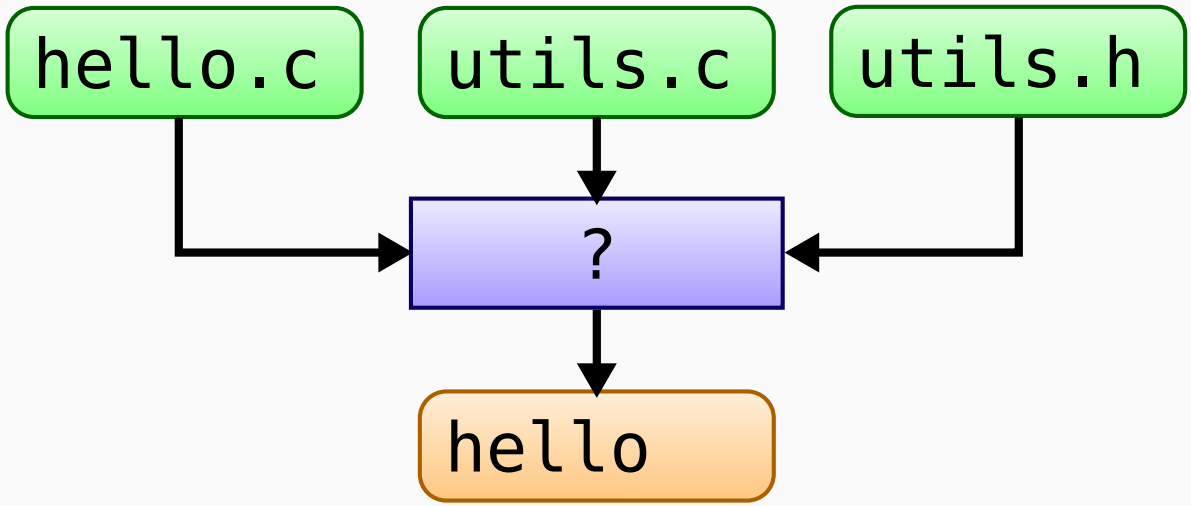
- Introduzione
- Shell script
- Make
- Autotools
- CMake
- Conclusioni

Introduzione

Processo di build

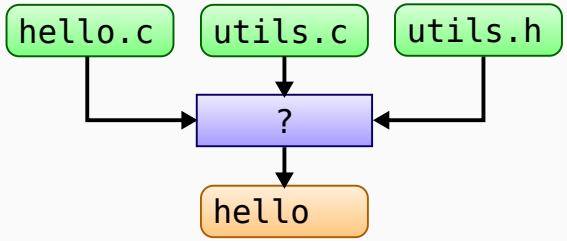


Processo di build




```
gcc -o hello hello.c utils.c
```

Processo di build



Processo di build:

- Input: il lavoro di un umano
- Output: un programma, una libreria, un documento, ...

Esempi:

- Codice C / C++ $\rightarrow$ un eseguibile o una libreria
- File di testo $\rightarrow$ Un documento PDF

Dove andrà il tuo software domani?

- Continuous integration / continuous delivery
  - Jenkins, Travis CI...
  - Server locale o “cloud”

Dove andrà il tuo software domani?

- Continuous integration / continuous delivery
  - Jenkins, Travis CI...
  - Server locale o “cloud”
- Distribuzioni: dpkg, rpm...

Dove andrà il tuo software domani?

- Continuous integration / continuous delivery
  - Jenkins, Travis CI...
  - Server locale o “cloud”
- Distribuzioni: dpkg, rpm...
- Sistemi embedded
  - Buildroot
  - Openembedded
  - OpenWRT

Dove andrà il tuo software domani?

- Continuous integration / continuous delivery
  - Jenkins, Travis CI...
  - Server locale o “cloud”
- Distribuzioni: dpkg, rpm...
- Sistemi embedded
  - Buildroot
  - Openembedded
  - OpenWRT
- Windows, MacOS, Android, iOS

Come vuoi compilare il tuo software?

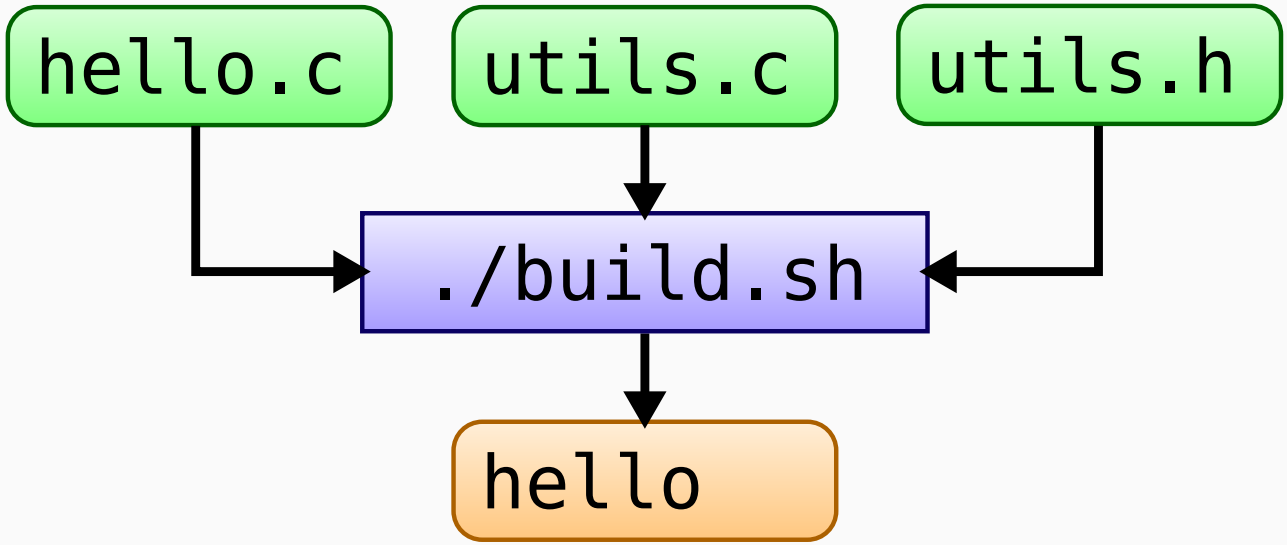
- Debug / Release
- `-Wall -Werror`
- Librerie: shared / static
- ...

Build automation: necessità

- Da riga di comando
- Deve funzionare in ambienti diversi
- Non deve avere *policy* hard-coded

Shell script

Shell script semplice



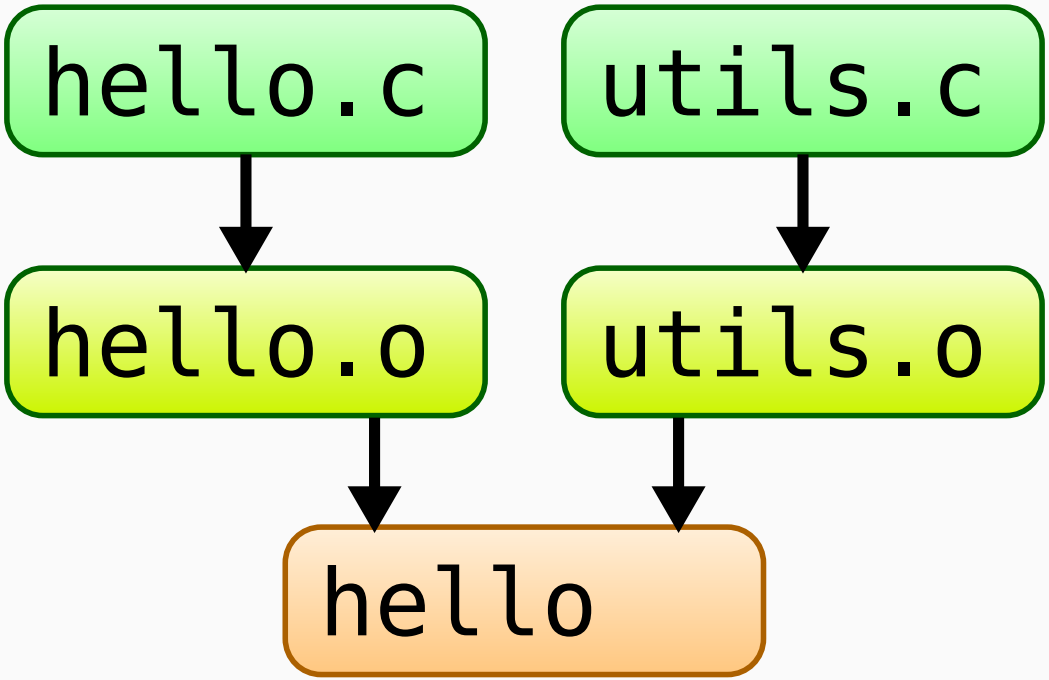
build.sh:

```
#!/bin/bash


gcc -o hello hello.c utils.c
```

Shell script semplice

- ✓ Funziona per casi molto semplici
- × Ricompila tutto ogni volta (se ci sono 100 file sorgente?)

File oggetto intermedi



File oggetto intermedi

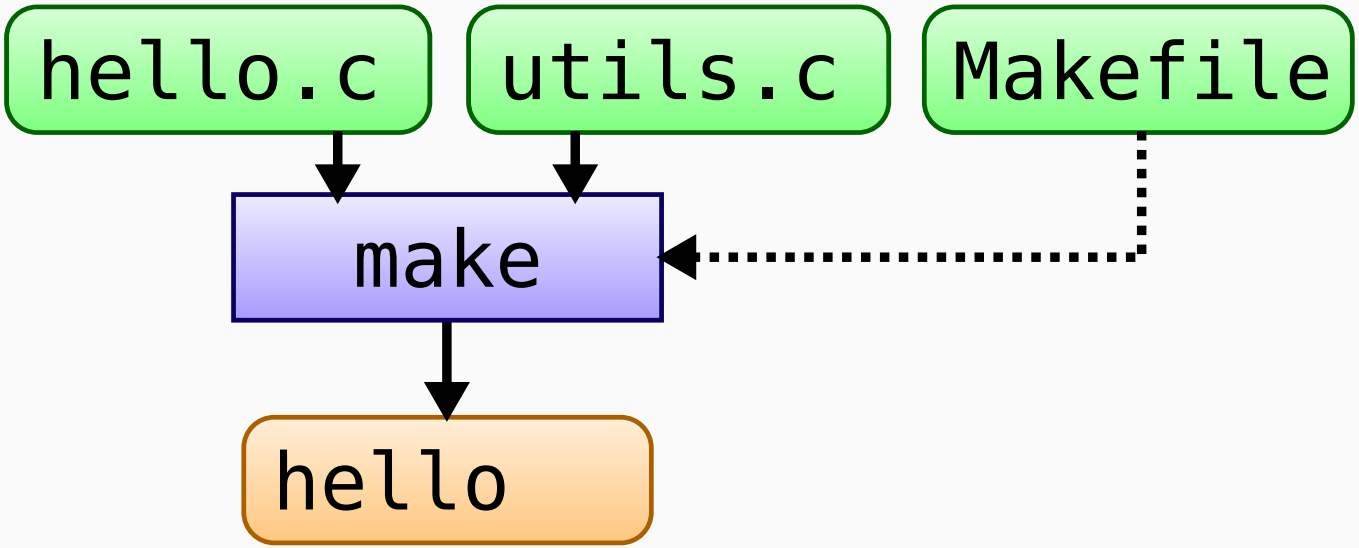
```
#!/bin/bash

if test hello.c -nt hello.o; then
    gcc -c hello.c
fi

if test utils.c -nt utils.o; then
    gcc -c utils.c
fi

if test hello.o -nt hello || test utils.o -nt hello; then
    gcc -o hello hello.c utils.c
fi
```

File oggetto intermedi

- ✓ Più efficiente
- × Scomodo da scrivere e mantenere
- × Non sfrutta CPU multi core

Make

Make



Makefile

`Makefile` contiene *regole*:

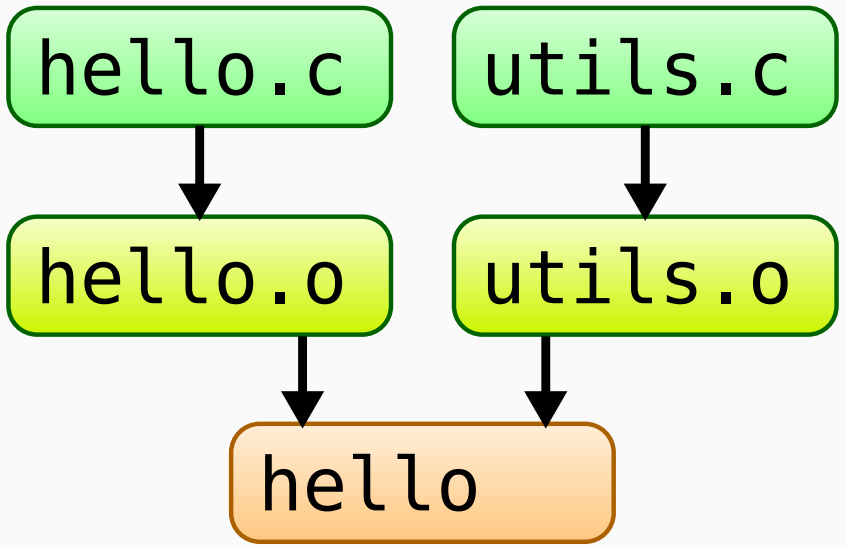
```
hello: hello.o utils.o
	gcc -o hello hello.o utils.o

hello.o: hello.c
	gcc -c hello.c

utils.o: utils.c
	gcc -c utils.c
```

Regole

```
hello: hello.o utils.o
        gcc -o hello hello.o utils.o
```

- `hello` = target
- `hello.o utils.o` = dipendenze
- `gcc ...` = comandi

Grafo delle dipendenze



Uso di Make

```
$ ls -tr
utils.h  utils.c  hello.c  Makefile
$ make
gcc -c hello.c
gcc -c utils.c
gcc -o hello hello.o utils.o
$ ls -tr
utils.h  utils.c  hello.c  Makefile  utils.o  hello.o  hel
$ touch utils.c
$ make
gcc -c utils.c
gcc -o hello hello.o utils.o
$
```

Altre regole utili

```
install: hello
    install -m 755 hello /usr/bin/

clean:
    rm -f hello hello.o utils.o

dist:
    tar czf hello.tar.gz utils.h utils.c hello.c Makefile
```

Alcuni vantaggi di Make

- Scrittura delle regole
  - Wildcard (`%.o: %.c`)
  - Implicit rules: varie regole predefinite (C, C++, ...)
  - ...
- Utilizzo
  - Build in parallelo (`make -j<N>`)
  - Build di un target specifico (`make utils.o`)

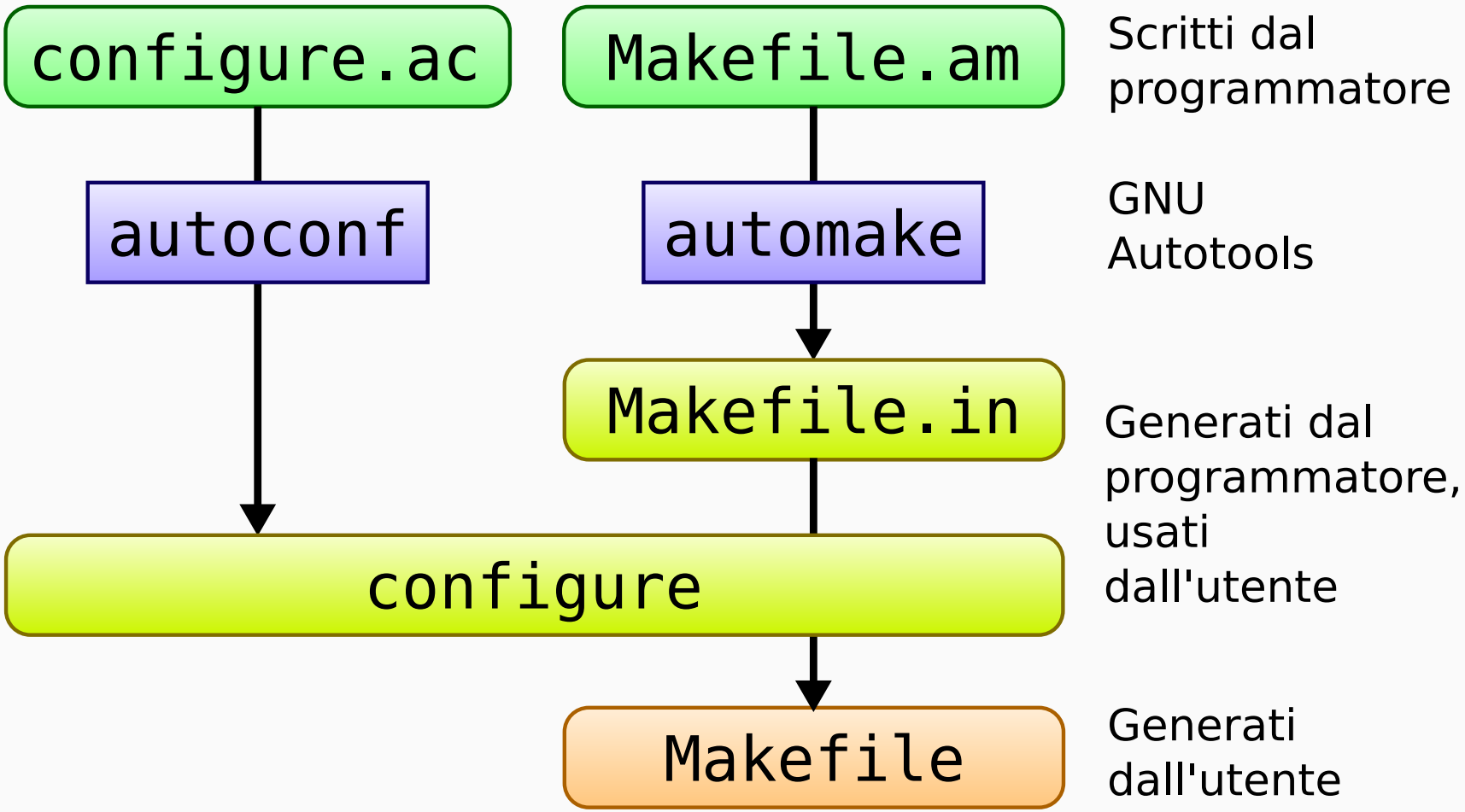
Cosa manca a Make

- Ricerca di librerie
- Prefix (`/usr`, `/usr/bin`, ...)
- Diverted installation (`DESTDIR`)
- Out of tree build
- Cross-compilazione (possibile ma scomoda)

Autotools

```
./configure
make
sudo make install
```

Autotools: schema semplificato

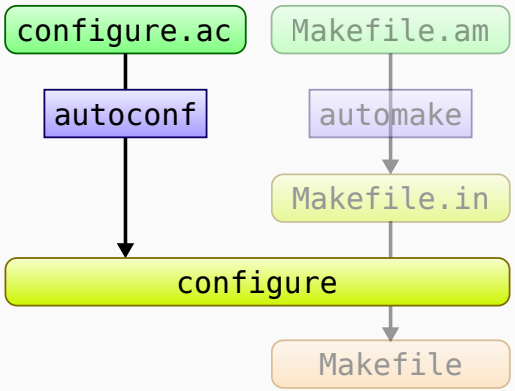


Autoconf

`configure.ac`

```
AC_INIT([hello], [1.0])
AC_OUTPUT
```

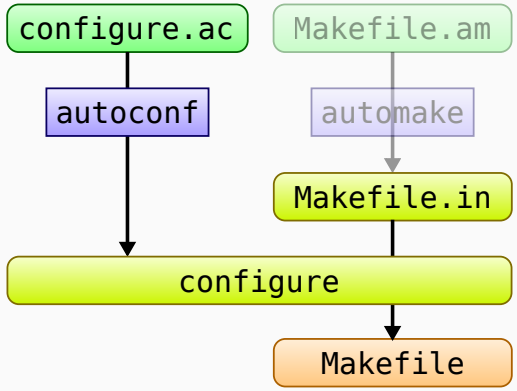
- È uno script di shell
- Con l'aggiunta di macro `m4`



Autoconf

configure.ac

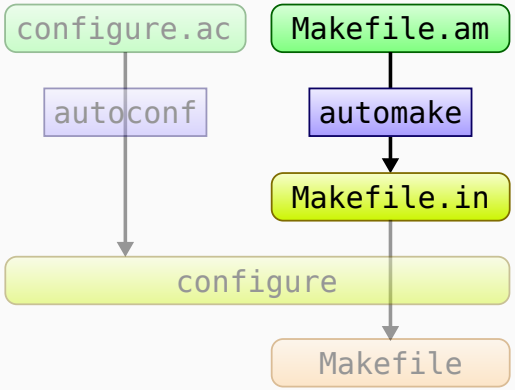
```
AC_INIT([hello], [1.0])
AC_PREREQ([2.69])
AC_PROG_CC
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```



Makefile.in

```
hello: hello.o utils.o
        @CC@ -o hello hello.o utils.o

%.o: %.c
        @CC@ -c $<
```

Automake



`Makefile.am`

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c utils.c utils.h
```

- È un normale Makefile
- Con l'aggiunta di costrutti di `automake`

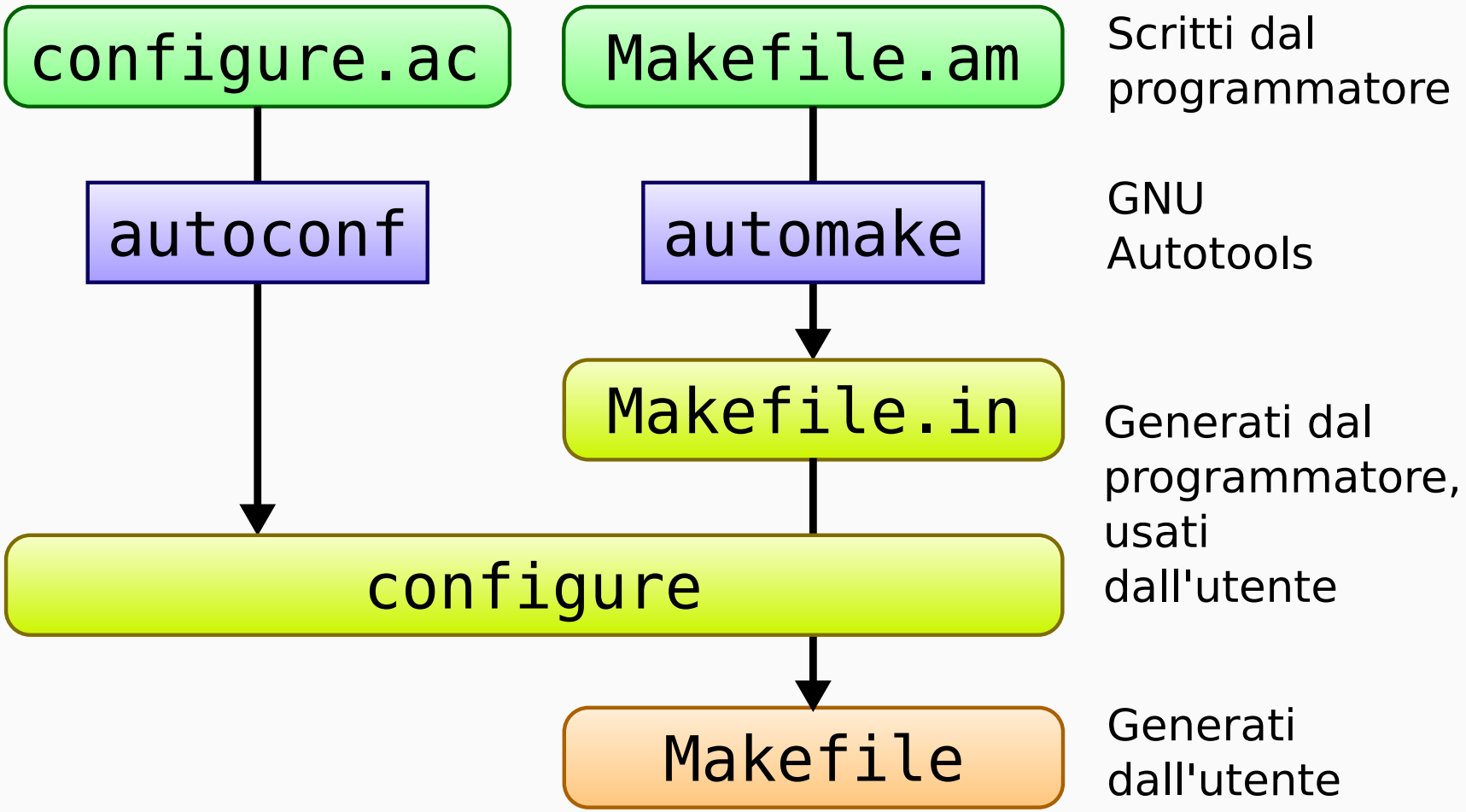
Automake: PSV e PLV

`Makefile.am` si compone di:

- `bin_PROGRAMS = hello`
  - Product List Variable (PLV)
  - Elenca i *prodotti* da generare

- `hello_SOURCES = hello.c utils.c utils.h`
  - Product Source Variable (PSV)
  - Elenca i *sorgenti* necessari per generare un prodotto

Autotools


configure.ac
Makefile.am
Scritti dal programmatore
autoconf
automake
GNU Autotools
Makefile.in
Generati dal programmatore, usati dall'utente
configure
Makefile
Generati dall'utente

Autotools: vantaggi

- Usato da moltissimi software
- Compatibilità con ogni sistema UNIX-like
- Moltissime macro già pronte
- Usano shell e Make
- Generano `Makefile`:
  - Conformi agli standard UNIX
  - Completi (`make install`, `make clean`, `make dist`, …)
- Vedi slide "Cosa manca a Make":
  - Ricerca di librerie
  - Prefix (`--prefix=/usr`)
  - Diverted installation (`DESTDIR`)
  - Out of tree build
  - Cross-compilazione con strumenti standard

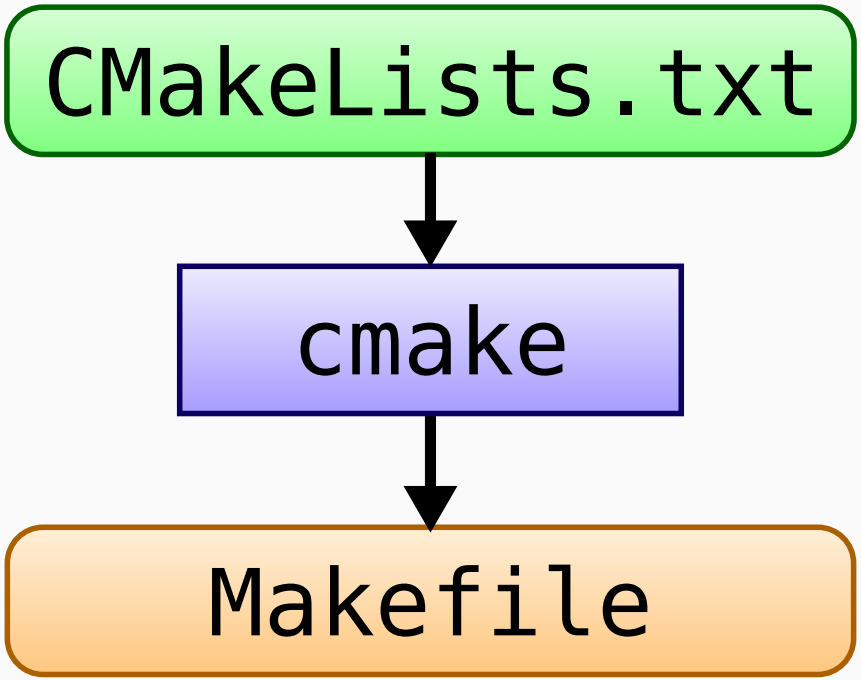
Autotools: svantaggi

- Praticamente inutilizzabili su sistemi non UNIX-like
- Usano shell e Make
- Complesso da imparare inizialmente
  - È facile scrivere codice "sbagliato" se non è chiaro il come funzionano
- `./configure` è lento (esecuzione sequenziale)

CMake

```
ccmake .
make
sudo make install
```

CMake

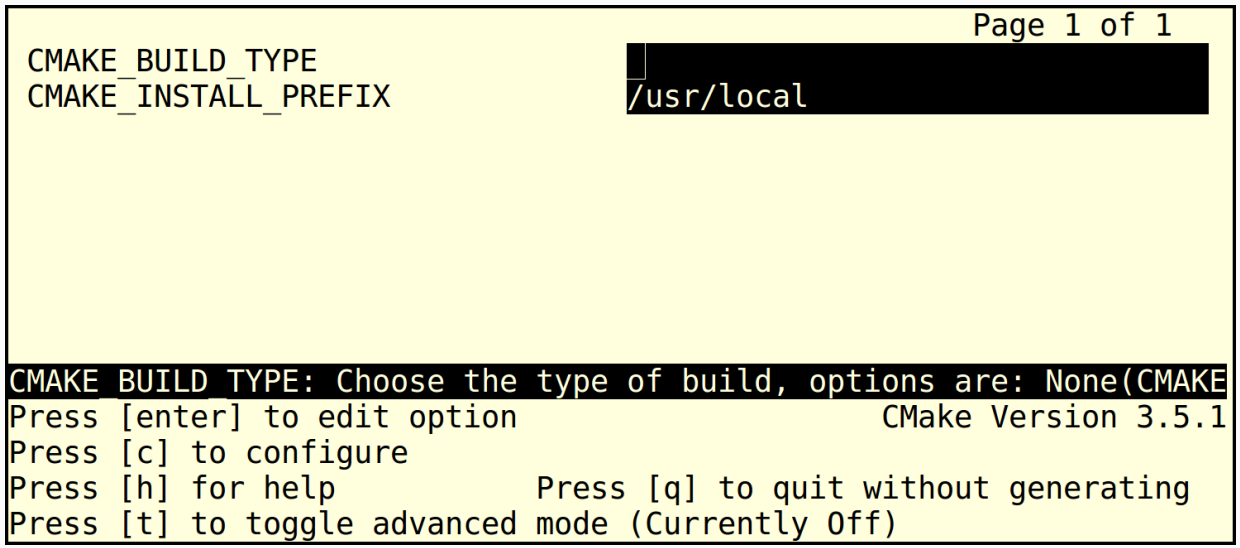


CMakeLists.txt semplice

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.6)
project(hello)
add_executable(hello hello.c utils.c)
```

ccmake


Page 1 of 1
CMAKE_BUILD_TYPE
CMAKE_INSTALL_PREFIX
/usr/local
CMAKE_BUILD_TYPE: Choose the type of build, options are: None(CMAKE
Press [enter] to edit option
CMake Version 3.5.1
Press [c] to configure
Press [h] for help
Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)

Installazione

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.6)
project(hello)
add_executable(hello hello.c utils.c)
install(TARGETS hello DESTINATION bin)
```

CMake: vantaggi

- Funziona su Linux, Windows, MacOS...
- Diffusione crescente
- Apprendimento iniziale facile
- `ccmake` e `cmake-gui` per configurazione interattiva
- Genera `Makefile`, progetti Visual Studio, XCode...
- Vedi slide "Cosa manca a Make":
  - Ricerca di librerie
  - Prefix (`CMAKE_INSTALL_PREFIX`)
  - Diverted installation (`DESTDIR`)
  - Out of tree build
  - Cross-compilazione

CMake: svantaggi

- Sintassi “non convenzionale”
- Non è banale gestire software complessi in modo corretto
- Minore diffusione

Conclusioni

Conclusioni

- Shell: non adatto
- Make: ottimo strumento, ma scrivere un ottimo `Makefile` è un lavoro enorme
- Autotools e CMake: strumenti completi e standard
  - Fanno molto più di quanto abbiamo visto
  - Esistono anche altri strumenti simili, meno diffusi

Quale strumento utilizzare?

Priorità di scelta *indicativa*

1. **Autotools o CMake**: quello che si conosce già
2. **CMake**: se serve compilare su Windows, MacOS...
3. **Autotools**: librerie di basso livello (sistema, interazione con il kernel)
4. **Make**: se Autotools e CMake non sono adatti
5. **Shell**: se non basta neanche Make...

Bibliografia

- GNU Make Manual (https://www.gnu.org/software/make/manual/)
- "A Practitioner's Guide to GNU Autoconf, Automake, and Libtool", John Calcote (https://www.nostarch.com/autotools.htm)
- "Autotools Mythbuster", Diego Elio "Flameeyes" Pettenò (https://autotools.io/index.html)
- "Autotools: A Demystification Tutorial", Thomas Petazzoni (https://elinux.org/images/4/43/Petazzoni.pdf, https://www.youtube.com/watch?v=_zX8LJ9Xjyk)
- CMake Tutorial: https://cmake.org/cmake-tutorial/
- CMake documentation: https://cmake.org/documentation/

Grazie per l'attenzione!

Domande?

luca@lucaceresoli.net

http://lucaceresoli.net