

Sestavení programů, Make

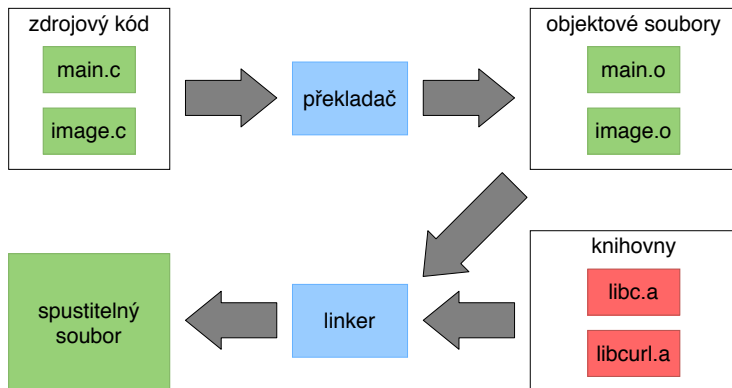
Michal Wiglasz, Jan Doležal, Viktor Malík, **Petr Veigend**

Fakulta informačních technologií Vysokého učení technického v Brně,
Božetěchova 1/2, 612 00 Brno – Královo Pole
veigend@fit.vut.cz



18. března 2026

- odevzdání plánů do **19.3.2026, 7:00**

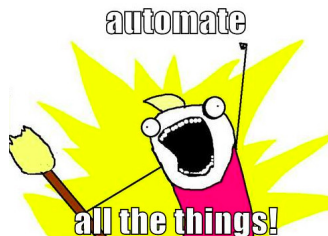


Proces sestavení programu je rozumné zautomatizovat:

- stejně jako jakýkoliv opakující se proces
- protože je to rychlejší, spolehlivější a méně otravné

Spousta nástrojů:

- C/C++ - Make, CMake...
- Java - Ant, Maven, Gradle...
- JS - Grunt, Gulp, Yarn
- (A to vše třeba na nějakém CI ¹ serveru a napojené na GIT - Jenkins, Travis)



¹continuous integration

- Automatizace překladu (nejen) programů
- Nezávislý na jazyku, multiplatformní, prověřený časem (1977)
- Detekuje změny a přeloží pouze změněné části
- Pravidla v souboru `GNUmakefile`, `makefile`, **Makefile**

Makefile se skládá z:

- cílů (targets)
- závislostí (prerequisites)
- příkazů pro vytvoření cílů ze závislostí (recipes)

`make [cíl1] [cíl2]`

- vykoná `cíl1` a `cíl2` (výchozí je první cíl zapsaný v Makefile)

`make --always-make [cíl] / make -B [cíl]`

- vykoná vše, tj. přeloží i nezměněné soubory

`make --just-print [cíl] / make --dry-run [cíl] / make -n [cíl]`

- pouze vypíše, co by se dělo, ale nic reálně nespustí

`make --jobs [počet] [cíl] / make -j [počet] [cíl]`

- spouští příkazy paralelně (až daný počet příkazů, případně bez omezení)

`make --print-data-base [cíl] / make -p [cíl]`

- navíc vypíše proměnné pravidla...

`make --makefile rules.mk [cíl] / make -f rules.mk [cíl]`

- použije pravidla z jiného souboru než Makefile

`make --debug / make -d`

- ladění Makefile, u `--debug` lze použít další parametry zápisem `--debug=(parametry)`
- např. `--debug=a` kompletní ladící informace, `--debug=b` základní ladící informace

Obsahuje definice cílů, závislostí a příkazů

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```

Obsahuje definice cílů, závislostí a příkazů

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```

■ cíle

Obsahuje definice cílů, závislostí a příkazů

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```

■ cíle ■ závislosti

Obsahuje definice cílů, závislostí a příkazů

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```

■ cíle ■ závislosti ■ příkazy

Obsahuje definice cílů, závislostí a příkazů

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```

■ cíle ■ závislosti ■ příkazy

```
1 # Obecně:
2 cíl1: závislost1 závislost2 ...
3     příkaz1
4     příkaz2
```

Obsahuje definice cílů, závislostí a příkazů

```
1 # MyProject Makefile
2 all: main.o
3 ____gcc -o myproject main.o
4
5 main.o: main.c
6 ____gcc -c main.c
```

■ cíle ■ závislosti ■ příkazy

```
1 # Obecně:
2 cíl1: závislost1 závislost2 ...
3 ____příkaz1
4 ____příkaz2
```

Příkaz musí být odsazen pomocí **tabulátoru**!

Co se stane po spuštění make?

- 1 Pokud není zadán cíl, najdi první (typicky `all`)
- 2 Aktualizuj jeho závislosti
- 3 Pokud se alespoň jedna závislost změnila, vykonej uvedené příkazy

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```

Co se stane po spuštění make?

- ❶ Pokud není zadán cíl, najdi první (typicky `all`)
- ❷ Aktualizuj jeho závislosti
- ❸ Pokud se alespoň jedna závislost změnila, vykonej uvedené příkazy

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```

Co se stane po spuštění make?

- 1 Pokud není zadán cíl, najdi první (typicky `all`)
- 2 Aktualizuj jeho závislosti
- 3 Pokud se alespoň jedna závislost změnila, vykonej uvedené příkazy

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```

Co se stane po spuštění make?

- 1 Pokud není zadán cíl, najdi první (typicky `all`)
- 2 Aktualizuj jeho závislosti
- 3 Pokud se alespoň jedna závislost změnila, vykonej uvedené příkazy

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```



Co se stane po spuštění make?

- ❶ Pokud není zadán cíl, najdi první (typicky `all`)
- ❷ Aktualizuj jeho závislosti
- ❸ Pokud se alespoň jedna závislost změnila, vykonej uvedené příkazy

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```

Co se stane po spuštění make?

- ❶ Pokud není zadán cíl, najdi první (typicky `all`)
- ❷ Aktualizuj jeho závislosti
- ❸ Pokud se alespoň jedna závislost změnila, vykonej uvedené příkazy

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```

Co se stane po spuštění make?

- 1 Pokud není zadán cíl, najdi první (typicky `all`)
- 2 Aktualizuj jeho závislosti
- 3 Pokud se alespoň jedna závislost změnila, vykonej uvedené příkazy

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```



Co se stane po spuštění make?

- ❶ Pokud není zadán cíl, najdi první (typicky `all`)
- ❷ Aktualizuj jeho závislosti
- ❸ Pokud se alespoň jedna závislost změnila, vykonej uvedené příkazy

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```

Co se stane po spuštění make?

- ❶ Pokud není zadán cíl, najdi první (typicky `all`)
- ❷ Aktualizuj jeho závislosti
- ❸ Pokud se alespoň jedna závislost změnila, vykonej uvedené příkazy

```
1 # MyProject Makefile
2 all: main.o
3     gcc -o myproject main.o
4
5 main.o: main.c
6     gcc -c main.c
```

Každý řádek příkazů se spouští v **samostatném shellu**

- případné komentáře make neřeší – celý řádek předá shellu
- změna adresáře, proměnných apod. se **nepřenes**e do dalšího příkazu
 - je nutné je zřetězit

Smaže soubory *.a v aktuální složce:

```
clean-libs:
    cd libs
    rm -f *.a
```

Smaže soubory *.a v podsložce libs:

```
clean-libs:
    cd libs && rm -f *.a
```

```
1 all: hello
2
3 hello: main.o util.o
4     gcc -g -Wall -std=c11 -o hello main.o util.o
5
6 main.o: main.c
7     gcc -g -Wall -std=c11 -c main.c
8
9 util.o: util.c
10    gcc -g -Wall -std=c11 -c util.c
11
12 clean:
13    rm -f hello *.o
```

```
1 all: hello
2
3 hello: main.o util.o
4     gcc -g -Wall -std=c11 -o hello main.o util.o
5
6 main.o: main.c
7     gcc -g -Wall -std=c11 -c main.c
8
9 util.o: util.c
10    gcc -g -Wall -std=c11 -c util.c
11
12 clean:
13    rm -f hello *.o
```

■ cíle

```
1 all: hello
2
3 hello: main.o util.o
4     gcc -g -Wall -std=c11 -o hello main.o util.o
5
6 main.o: main.c
7     gcc -g -Wall -std=c11 -c main.c
8
9 util.o: util.c
10    gcc -g -Wall -std=c11 -c util.c
11
12 clean:
13    rm -f hello *.o
```

■ cíle ■ závislosti

```
1 all: hello
2
3 hello: main.o util.o
4     gcc -g -Wall -std=c11 -o hello main.o util.o
5
6 main.o: main.c
7     gcc -g -Wall -std=c11 -c main.c
8
9 util.o: util.c
10    gcc -g -Wall -std=c11 -c util.c
11
12 clean:
13    rm -f hello *.o
```

■ cíle ■ závislosti ■ příkazy

PROMĚNNÉ UNIVERZÁLNÍ A IMPLICITNÍ PRAVIDLA

```
CC = gcc
CFLAGS = -Wall --std=c11
CXX = g++
CXXFLAGS = -Wall --std=c++11
LDLIBS = -lzip -lz

EXECUTABLE = hello
OBJS = main.o util.o log.o
```

```
# konkatenace
CFLAGS += -O3
```

Proměnné se dá použít prakticky kdekoliv:

```
all: $(EXECUTABLE)
$(EXECUTABLE): $(OBJS)
    $(CC) $(FLAGS) -o $@ $^ $(LDLIBS)
```

Automatické proměnné

\$@	Cíl
\$<	První závislost
\$?	Všechny závislosti novější než cíl
\$^	Všechny závislosti bez duplicit
\$+	Všechny závislosti vč. duplicit
\$(@D) \$(<D) \$(^D) \$(+D)	Jména složek
\$(@F) \$(<F) \$(^F) \$(+F)	Jména souborů

Obsah proměnné se vyhodnocuje až při použití:

```
CFLAGS = $(include_dirs)
include_dirs = -Ifoo
```

Ale pozor na rekurzi:

```
CFLAGS = $(CFLAGS) -Ibar
```

Operátor `:=` vyhodnotí obsah proměnné ihned:

```
CFLAGS := $(CFLAGS) -Ibar
```

```
1 all: hello
2
3 hello: main.o util.o
4     gcc -g -Wall -std=c11 -o hello main.o util.o
5
6 main.o: main.c
7     gcc -g -Wall -std=c11 -c main.c
8
9 util.o: util.c
10    gcc -g -Wall -std=c11 -c util.c
11
12 clean:
13    rm -f hello *.o
```

```
1 EXECUTABLE = hello
2 CC = gcc
3 CFLAGS = -g -Wall --std=c11
4
5 all: hello
6
7 hello: main.o util.o
8     gcc -g -Wall -std=c11 -o hello main.o util.o
9
10 main.o: main.c
11     gcc -g -Wall -std=c11 -c main.c
12
13 util.o: util.c
14     gcc -g -Wall -std=c11 -c util.c
15
16 clean:
17     rm -f hello *.o
```

```
1 EXECUTABLE = hello
2 CC = gcc
3 CFLAGS = -g -Wall --std=c11
4
5 all: $(EXECUTABLE)
6
7 $(EXECUTABLE): main.o util.o
8     $(CC) $(CFLAGS) -o $@ $^
9
10 main.o: main.c
11     $(CC) $(CFLAGS) -c $^
12
13 util.o: util.c
14     $(CC) $(CFLAGS) -c $^
15
16 clean:
17     rm -f $(EXECUTABLE) *.o
```

```
1 EXECUTABLE = hello
2 CC = gcc
3 CFLAGS = -g -Wall --std=c11
4
5 all: $(EXECUTABLE)
6
7 $(EXECUTABLE): main.o util.o
8     $(CC) $(CFLAGS) -o $@ $^
9
10 main.o: main.c
11     $(CC) $(CFLAGS) -c $^
12
13 util.o: util.c
14     $(CC) $(CFLAGS) -c $^
15
16 clean:
17     rm -f $(EXECUTABLE) *.o
```



tato pravidla jsou de facto stejná

Lze psát univerzální pravidla pomocí znaku %

```
%.o: %.c  
    $(CC) $(CFLAGS) -c $<
```

Make má v sobě zabudovanou sadu implicitních pravidel:

```
%.c → %.o  
    $(CC) $(CPPFLAGS) $(CFLAGS) -c
```

```
%.c, %.cpp, %.C → %.o  
    $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c
```

```
%.o → %  
    $(CC) $(LDFLAGS) n.o $(LOADLIBES) $(LDLIBS)
```

```
1 EXECUTABLE = hello
2 CC = gcc
3 CFLAGS = -g -Wall --std=c11
4
5 all: $(EXECUTABLE)
6
7 $(EXECUTABLE): main.o util.o
8     $(CC) $(CFLAGS) -o $@ $^
9
10 main.o: main.c
11     $(CC) $(CFLAGS) -c $^
12
13 util.o: util.c
14     $(CC) $(CFLAGS) -c $^
15
16 clean:
17     rm -f $(EXECUTABLE) *.o
```

```
1 EXECUTABLE = hello
2 CC = gcc
3 CFLAGS = -g -Wall --std=c11
4
5 all: $(EXECUTABLE)
6
7 $(EXECUTABLE): main.o util.o
8     $(CC) $(CFLAGS) -o $@ $^
9
10 %.o: %.c
11     $(CC) $(CFLAGS) -c $^
12
13
14
15
16 clean:
17     rm -f $(EXECUTABLE) *.o
```

```
1 EXECUTABLE = hello
2 CC = gcc
3 CFLAGS = -g -Wall --std=c11
4
5 all: $(EXECUTABLE)
6
7 $(EXECUTABLE): main.o util.o
8
9
10
11
12 # s využitím zabudovaných pravidel není třeba psát skoro nic
13
14
15
16 clean:
17     rm -f $(EXECUTABLE) *.o
```

```
1
2
3
4
5 all: hello
6
7 hello: main.o util.o
8     gcc -g -Wall -std=c11 -o hello main.o util.o
9
10 main.o: main.c
11     gcc -g -Wall -std=c11 -c main.c
12
13 util.o: util.c
14     gcc -g -Wall -std=c11 -c util.c
15
16 clean:
17     rm -f hello *.o
```

```
1 EXECUTABLE = hello
2 CC = gcc
3 CFLAGS = -g -Wall --std=c11
4
5 all: $(EXECUTABLE)
6
7 $(EXECUTABLE): main.o util.o
8
9
10
11
12
13
14
15
16 clean:
17     rm -f $(EXECUTABLE) *.o
```

DALŠÍ VYCHYTÁVKY

Volání funkcí:

```
$(function arg1,arg2,arg3 ...)
```

Například:

```
SOURCES = $(wildcard *.c)
OBJS = $(patsubst %.c,%.o,$(SOURCES))

USERNAME := $(shell whoami)
HOSTNAME := $(shell hostname)
```

Program poskytující informace o nainstalovaných knihovnách, umožňuje automaticky nastavit překladač.

Použití v Makefile:

```
CFLAGS += $(shell pkg-config --cflags libzip)
LDLIBS += $(shell pkg-config --libs libzip)
```

Do CFLAGS přidá:

```
-I/usr/lib/libzip/include
```

Do LDLIBS přidá:

```
-lzip -lz
```

Speciální cíl `.PHONY` umožňuje definovat, které cíle se mají provést vždy bez ohledu na existenci souborů.

```
.PHONY: clean
```

```
clean:
```

```
    rm -f $(EXECUTABLE) *.o
```

Pokud by chyběla definice `.PHONY` a existoval by soubor `clean`, příkaz by se nevykonával.

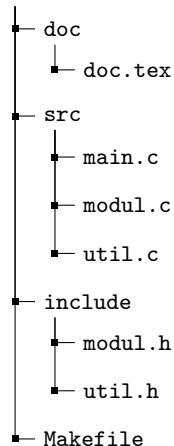
Říká, kde se mají hledat **závislosti** cílů.

```
# všechny typy souborů
VPATH = doc src include

# jen některé typy
vpath %.c src
vpath %.h include
vpath %.tex doc

modul.o: modul.c modul.h util.h
    $(CC) $(CFLAGS) -c $<
```

Projekt



gcc umí vygenerovat cíle a závislosti pro Makefile

```
gcc -MM *.c
```

Výstup je například:

```
dtsp.o: dtsp.c dtsp.h random.h config.h load.h  
load.o: load.c config.h load.h dtsp.h random.h  
main.o: main.c dtsp.h random.h config.h load.h
```

Lze použít přímo v Makefile (funkce `include`, `-MMD`).

```
1 LOGIN = xwigla00
2 SERVER = merlin.fit.vutbr.cz
3 SERVER_DIR = ~/KRY/proj1
4 ZIP_FILE = $(LOGIN).zip
5 .PHONY: all pack run clean upload
6
7 all: $(EXECUTABLE)
8 pack: $(ZIP_FILE)
9 run: $(EXECUTABLE)
10     ./$(EXECUTABLE) $(ARGS)
11 $(EXECUTABLE): $(OBJS)
12 clean:
13     rm -f $(EXECUTABLE) *.o $(ZIP_FILE)
14 $(ZIP_FILE): *.c *.h Makefile doc/doc.pdf
15     zip -j $@ $^
16 upload: $(ZIP_FILE)
17     scp $^ $(SERVER):$(SERVER_DIR)
18     ssh $(LOGIN)@$(SERVER) \
19         "cd $(SERVER_DIR) && unzip $^ && make"
```

```
valgrind: $(EXECUTABLE)
    valgrind ./$(EXECUTABLE) $(ARGS)

gdb: $(EXECUTABLE)
    gdb -ex ./$(EXECUTABLE) --args $(ARGS)

leaks: $(EXECUTABLE)
    valgrind --track-origins=yes --leak-check=full \
        --show-reachable=yes ./$(EXECUTABLE) $(ARGS)
```

```
valgrind: $(EXECUTABLE)
    valgrind ./$(EXECUTABLE) $(ARGS)

gdb: $(EXECUTABLE)
    gdb -ex ./$(EXECUTABLE) --args $(ARGS)

leaks: $(EXECUTABLE)
    valgrind --track-origins=yes --leak-check=full \
        --show-reachable=yes ./$(EXECUTABLE) $(ARGS)
```

Proměnná lze nastavit i při spuštění programu make.

```
$ make gdb ARGS='--input /foo/bar'
```

- Presentace – Michal Wiglasz, Jan Doležal, Viktor Malík
- https://www.gnu.org/software/make/manual/html_node/
- <http://make.mad-scientist.net/papers/rules-of-makefiles/>
- <https://www.cmcrossroads.com/article/basics-vpath-and-vpath>
- <http://make.mad-scientist.net/papers/how-not-to-use-vpath/>
- <https://stackoverflow.com/questions/1745939/debugging-gnu-make>

Děkuji za pozornost