Wprowadzenie do `make`

Cel wykładu

Zapoznanie z programem `make` i zasadami jego wykorzystania do zarządzania projektami złożonymi z wielu plików.

Wprowadzenie

1. Przeznaczenie `make`: organizacja wielu plików źródłowych w jeden projekt; umożliwienie automatycznej propagacji zmian w plikach źródłowych do finalnego kodu;

2. Organizacja: `makefile` – „program" opisujący które pliki zależą od których i w jaki sposób;

3. Przykład:

   Wyobraźmy sobie, że chcemy zgrupować pewne użyteczne funkcje w jeden plik, skompilować je i używać już w postaci skompilowanej przez pozostały okres studiów. Możemy np. utworzyć z tych funkcji plik `utils.c` (i oczywiście stosowny plik `utils.h`), skompilować go:

   ```
   cc -c -o utils.o utils.c
   ```

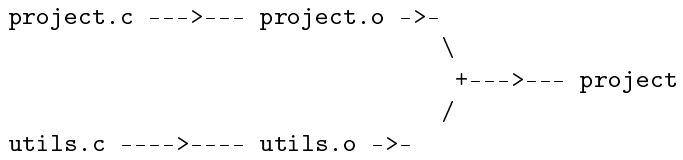
   Jeżeli potem będziemy chcieli używać funkcji z biblioteki `utils`, powiedzmy, że w programie zawartym w pliku `project.c`, to możemy skompilować go następująco

   ```
   cc -c -o project.o project.c
   cc -o project project.o utils.o
   ```

   lub

   ```
   cc -o project project.c utils.o
   ```

   Można przedstawić zależności między plikami projektu w formie następującego diagramu. Zmiany w treści plików powinny być propagowane w projekcie zgodnie z kierunkiem strzałek

4. Jeżeli projekt składałby się z większej ilości plików, to trudno śledzić wszystkie powiązania – do tego właśnie służy `make`.

5. Dla naszego przykładowego projektu możemy utworzyć plik opisujący przedstawione powyżej powiązania. Program `make` czyta taki plik i uaktualnia wszystkie pliki projektu tak, aby nie były one starsze niż pliki, od których zależą.

   Plik opisujący nazywamy gwarowo *makefile* i zwykle też nosi on taką nazwę: `makefile` lub `Makefile`.

Struktura `makefile`

`Makefile`może składać się z:

1. reguł prostych (lub *explicite*) (ang. *explicit rules*)
2. reguł domyślnych (ang. *implicit rules*)
3. definicji zmiennych
4. dyrektyw: `include`, `if`, `define`
5. komentarzy

Przykładowy `makefile` dla opisanego powyżej projektu może wyglądać jak następuje:

```
project: project.o utils.o
   cc -o project project.o utils.o

project.o: project.c
   cc -c -o project.o project.c

utils.o: utils.c
   cc -c -o utils.o utils.c
```

Plik ten skalda się z samych reguł prostych. Rozważmy np. regułę

```
project: project.o utils.o
   cc -o project project.o utils.o
```

Składa się ona z: *celu* (ang. *target*) (tu `project`) zakończonego dwukropkiem. Po dwukropku występuje lista *zależności* (ang. *depedencies*) to znaczy plików, od których zależy cel. W następnej linii rozpoczynają się *komendy* (ang. *commands*), które prowadzą od zależności do celu. Uwaga: komendy są poprzedzane tabulatorem a nie spacjami!

Wywołanie programu `make`:

```
make project
```

Spowoduje sprawdzenie dat ostatnich modyfikacji wszystkich plików i, w razie takiej potrzeby, przebudowanie projektu.

Zapomnieliśmy tu jeszcze o pliku `utils.h`, od którego zależą oczywiście zarówno `utils.c` jak i `project.c`. Stosowne modyfikacje wprowadzimy do kolejnej wersji `makefile`:

```
# nasz przykladowy makefile

HEADERS = utils.h
project: project.o utils.o
   cc -o project project.o utils.o

project.o: project.c $(HEADERS)
   cc -c -o project.o project.c

utils.o: utils.c $(HEADERS)
   cc -c -o utils.o utils.c
```

W pliku tym pokazano dodatkowo wykorzystanie zmiennej (`HEADERS`) i komentarz.

Cele puste

Przyjrzyjmy się innemu plikowi `makefile`:

```
all: build install

build: myprog yourprog

myprog: main.o utils.o
   cc -o myprog main.o utils.o

main.o: main.c utils.h
   cc -c -o main.o main.c

utils.o: utils.c utils.h
   cc -c -o utils.o utils.c

yourprog: yourmain.o utils.o
   cc -o yourprog yourmain.o utils.o

install:
      cp myprog /usr/local/bin/
      cp yourprog /usr/share/bin/
```

W pliku tym występują *cele puste* (ang. *dummy targets* lub *phony targets*), które nie są nazwami plików. Służą one tylko do organizacji celów w struktury.

Więcej o komendach

Czasem aby zbudować jakiś cel należy wykonać więcej niż jedną komendę. `make`dopuszcza taką możliwość, np:

```
   projekty: formularz.tex ${TEMATY}  ../bib/projekty_c.bib
      latex formularz
      bibtex formularz
      latex formularz
      latex formularz
```

`make` przerywa działanie gdy jakaś komenda zwróci sygnał o wystąpieniu błędu. Możemy zabezpieczyć sie przed taką ewentualnością, np:

```
test:
   -mkdir ./test
   cp dane test/dane
   randomize test/dane
   prog < test/dane > test/wyniki
```

Znak "-" przed poleceniem `mkdir` zabezpiecza nas przed przerwaniem działania `make` w sytuacji, gdy podkatalog `test` juz istnieje.

Zmienne

Zmienne pozwalają parametryzować `makefile` tak, aby np. łatwiej je było zmieniać. Rozpatrzmy np. projekt

```
CC = /usr/bin/gcc
CFLAGS = -Wall

LIB = ar

BASEDIR = /usr/local
PROGDIR = $(BASEDIR)/bin
LIBDIR = $(BASEDIR)/lib

all: build install

build: myprog mylib

myprog: main.o utils.o
   $(CC) $(CFLAGS) -o myprog main.o utils.o

main.o: main.c utils.h
   $(CC) $(CFLAGS) -c -o main.o main.c

utils.o: utils.c utils.h
   $(CC) $(CFLAGS) -c -o utils.o utils.c

math.o: math.c math.h utils.h
   $(CC) $(CFLAGS) -c -o math.o math.c

mylib: utils.o math.o
   $(LIB) -c libmy.a utils.o math.o

install:
      cp myprog $(PROGDIR)
      cp mylib $(LIBDIR)
```

Wielopoziomowe `makefile`

Komenda `makefile` może rekursywnie wywoływać `make`. Można to wykorzystać przy tworzeniu projektów składających się z kilku części. Rozpatrzmy dla przykładu projekt, który składa się z kilku aplikacji: `prepro`, `sim` i `viewer`. Kod projektu będzie przechowywany w następującym drzewie katalogów:

```
app---+---prepro
      |
      +---sym
      |
      +---viewer
```

Odpowiednie pliki `makefile` zostaną umieszczone we wszystkich katalogach. Na przykład plik umieszczony w katalogu głównym (`app`) mógłby wyglądać następująco:

```
BINDIR = /usr/local/bin

all:
   ( cd prepro; make all )
   ( cd sim; make all )
   ( cd viewer; make all )

install:
   ( cd prepro; make install BINDIR=${BINDIR} )
   ( cd sim; make install BINDIR=${BINDIR} )
   ( cd viewer; make install BINDIR=${BINDIR} )

clean:
   ( cd prepro; make clean )
   ( cd sim; make clean )
   ( cd viewer; make clean )
```

Ten `makefile` jedynie uruchamia `make` w poszczególnych pod-katalogach. Plik `makefile` w katalogu `sim` mógłby wyglądać następująco:

```
BINDIR = /usr/bin

all: simulator

simulator: main.o fem.o reader.o
   cc -o simulator main.o fem.o reader.o

install:
   cp simulator ${BINDIR}

clean:
   -rm simulator *.o
```

GNU `make`

GNU `make` to szeroko rozpowszechniony, przenośny program *public domain* stanowiący jednocześnie bardzo dobry przykład `make`. Wyposażony w wiele bardzo wygodnych mechanizmów.

GNU `make` – komendy domyślne

Przeanalizujmy następujący przykład:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

Nie ma tu komend opisujących jak zrobić pliki "`.o`" z plików "`.c`". GNU`make` „wie" jak należy to zrobić. Tego typu domyślne reguły są pobierane z plików konfiguracyjnych.

GNU `make` – grupowanie w/g zależności

`makefile` analogiczny do poprzedniego możemy też zapisać następująco:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

Teraz jest ono pogrupowane według zależności a nie według celów.

GNU `make` – `.PHONY`

Jeżeli jakiś cel nie jest nazwą pliku to możemy to jawnie podkreślić, określając go jako `.PHONY`. Zabezpiecza to nas przed nieoczekiwanym działaniem `make` w sytuacji, gdyby w katalogu pojawił się plik o takiej nazwie.

GNU `make` – puste pliki-cele

Czasem korzystne jest stosowanie pustych plików-celów, które służą tylko jako flagi. Pokazuje to poniższy przykład, który wykorzystuje plik-flagę o nazwie `print` do zaznaczenia, które pliki źródłowe były już drukowane:

```
print: *.c
        lpr -p $?
        touch print
```

Trzeba jeszcze dodać, że automatyczna zmienna `$?` przechowuje wszystkie zależności, które są nowsze od celu.

GNU `make` – wbudowane cele specjalne

Pewne nazwy mają specjalne znaczenie, jeśli występują jako cele:
`.PHONY` – zależności tego celu to cele, których komendy będą wykonywane zawsze;
`.SUFFIXES` – lista przyrostków używanych przy poszukiwaniu reguł przyrostkowych;
`.DEFAULT` – komendy wyspecyfikowane po tym celu są wykonywane dla wszystkich celów, dla których nie znaleziono żadnych celów;
`.PRECIOUS` – jeżeli `make` zostanie zabite w trakcie robienia tak zadeklarowanego celu, to cel nie jest usuwany;
`.IGNORE` – ignorowane są błędy w komendach dla tak zadeklar. celu;
`.SILENT` – `make` nie pokazuje śladu wykonywania komend dla takiego celu;

GNU `make` – wildcards

```
print: *.c
        lpr -p $?
        touch print


clean:
        rm -f *.o
```

Rozwinięcie wildcards nie następuje przy podstawieniu do zmiennej, tzn.

```
objects = *.o
```

nada zmiennej `objects` wartość `"*.o"`. Rozwinięcie może natomiast wystąpić pożyciu takiej zmiennej w komendzie, np.:

```
rm $objects
```

Czasem dogodnie jest używać funkcji:

```
objects := $(wildcard *.o)
```

```
objects := $(pathsubst %.c,%.o,$(wildcard *.c))
```

GNU make – wiele celów w jednej linii

```
bigoutput littleoutput : text.g
        generate text.g -$(subst output,,$@) > $@
```

GNU make – reguły ze statycznym wzorcem

```
objects = foo.o bar.o

all: $(objects)

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

GNU make – użycie rekursywne

```
subsystem:
        cd subdir && $(MAKE)
```

```
subsystem:
        $(MAKE) -C subdir
```

GNU make – define

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c $@
endef
...
foo.c : foo.y
        $(run-yacc)
```

GNU make – zmienne rozwijane rekursywnie

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:;echo $(foo)
```

Wydrukuje `Huh?`

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

GNU make – zmienne rozwijane natychmiast

```
x := foo
y := $(x) bar
x := later
```

jest równoważne

```
y := foo bar
x := later
```

GNU make – zmienne definiowane warunkowo i dodawanie do zmiennych

```
FOO ?= bar
```

jest równoważne

```
ifeq ($(origin FOO), undefined)
  FOO = bar
endif
```

```
objects += inny.o

objects := $(objects) jeszcze_inny.o
```

GNU make – katalog reguł domyślnych

C `n.c` → `n.o` : `$(CC) -c $(CPPFLAGS) $(CFLAGS)`
C++ `n.cc` lub `n.C` → `n.o` : `$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)`
Pascal `n.p` → `n.o` : `$(PC) -c $(PFLAGS)`
Fortan (Ratfor)

Modula-2
Asembler
Linker
Yacc
Lex
TEX
TEXinfo i Info

GNU `make` – katalog zmiennych dla reguł domyślnych

```
AR          ARFLAGS
AS          ASFLAGS
CC          CFLAGS
CXX         CXXFLAGS
CPP         CPPFLAGS
FC          FFFLAGS
LEX         LFLAGS
PC          PFLAGS
YACC        YFLAGS
MAKEINFO
TEX
TEXI2DVI
WEAVE
```

GNU `make` – łańcuchy reguł domyślnych

Jeżeli plik można utworzyć z innego przy pomocy sekwencji reguł domyślnych, to GNU`make` tak właśnie zrobi. Powstające przy okazji pliki przejściowe są tworzone w razie potrzeby i usuwane po tym, jak już nie są potrzebne.

GNU `make` – reguły wzorcowe

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

Robi jeden object z wszystkich plików wejściowych

GNU `make` – podsumowanie

Wszystkie poleceniaGNU`make`:

`define` *variable*

`endef`

Define a multi-line, recursively-expanded variable. Sequences.

`ifdef` *variable*

`ifndef` *variable*

`ifeq` (*a*,*b*)

`ifeq` "*a*" "*b*"

`ifeq` '*a*' '*b*'

`ifneq` (*a*,*b*)

`ifneq` "*a*" "*b*"

`ifneq` '*a*' '*b*'

`else`

`endif`

Conditionally evaluate part of the makefile. Conditionals.

`include` *file*

Include another makefile. Include, ,Including Other Makefiles.

`override` *variable* = *value*

`override` *variable* := *value*

`override` *variable* += *value*

`override` define *variable*

`endef`

Define a variable, overriding any previous definition, even one from the command line. Override Directive, ,The `override` Directive.

`export`

Tell `make` to export all variables to child processes by default. Variables/Recursion, , Communicating Variables to a Sub-`make`.

`export` *variable*

`export` *variable* = *value*

`export` *variable* := *value*

`export` *variable* += *value*

`unexport` *variable* Tell `make` whether or not to export a particular variable to child processes. Variables/Recursion, , Communicating Variables to a Sub-`make`.

`vpath` *pattern path* Specify a search path for files matching a `%` pattern. × Selective Search, , The `vpath` Directive.

`vpath` *pattern* Remove all search paths previously specified for *pattern*.

`vpath` Remove all search paths previously specified in any `vpath` directive.

Funkcje do manipulacji tekstem

`$(subst \var{from},\var{to},\var{text})` Replace *from* with *to* in *text*. × Text Functions, , Functions for String Substitution and Analysis.

`$(patsubst \var{pattern},\var{replacement},\var{text})` Replace words matching *pattern* with *replacement* in *text*. Text Functions, , Functions for String Substitution and Analysis.

`$(strip \var{string})` Remove excess whitespace characters from *string*. Text Functions, , Functions for String Substitution and Analysis.

`$(findstring \var{find},\var{text})` Locate *find* in *text*. Text Functions, , Functions for String Substitution and Analysis.

`$(filter \var{pattern}\dots{},\var{text})` Select words in *text* that match one of the *pattern* words. Text Functions, , Functions for String Substitution and Analysis.

`$(filter-out \var{pattern}\dots{},\var{text})` Select words in *text* that *do not* match any of the *pattern* words. Text Functions, , Functions for String Substitution and Analysis.

`$(sort \var{list})` Sort the words in *list* lexicographically, removing duplicates. × Text Functions, , Functions for String Substitution and Analysis.

`$(dir \var{names}\dots{})` Extract the directory part of each file name. File Name Functions, ,Functions for File Names.

`$(notdir \var{names}\dots{})` Extract the non-directory part of each file name. File Name Functions, ,Functions for File Names.

`$(suffix \var{names}\dots{})` Extract the suffix (the last `.` and following characters) of each file name. File Name Functions, ,Functions for File Names.

`$(basename \var{names}\dots{})` Extract the base name (name without suffix) of each file name. File Name Functions, ,Functions for File Names.

`$(addsuffix \var{suffix},\var{names}\dots{})` Append *suffix* to each word in *names*. File Name Functions, ,Functions for File Names.

`$(addprefix \var{prefix},\var{names}\dots{})` Prepend *prefix* to each word in *names*. File Name Functions, ,Functions for File Names.

`$(join \var{list1},\var{list2})` Join two parallel lists of words. File Name Functions, ,Functions for File Names.

`$(word \var{n},\var{text})` Extract the *n*th word (one-origin) of *text*. File Name Functions, ,Functions for File Names.

`$(words \var{text})` Count the number of words in *text*. File Name Functions, ,Functions for File Names.

`$(firstword \var{names}\dots{})` Extract the first word of *names*. File Name Functions, ,Functions for File Names.

`$(wildcard \var{pattern}\dots{})` Find file names matching a shell file name pattern (*not* a `%` pattern). Wildcard Function, ,The Function `wildcard`.

`$(shell \var{command})`

Execute a shell command and return its output. Shell Function, , The `shell` Function.

`$(origin \var{variable})`

Return a string describing how the `make` variable *variable* was defined. Origin Function, , The `origin` Function.

`$(foreach \var{var},\var{words},\var{text})`

Evaluate *text* with *var* bound to each word in *words*, and concatenate the results. Foreach Function, ,The `foreach` Function.

Zmienne automatyczne:

`$@` The file name of the target.

`$%` The target member name, when the target is an archive member.

`$<` The name of the first dependency.

`$?` The names of all the dependencies that are newer than the target, with spaces between them. For dependencies which are archive members, only the member named is used (Archives).

`$^`

`$+` The names of all the dependencies, with spaces between them. For dependencies which are archive members, only the member named is used (Archives). The value of `$^` omits duplicate dependencies, while `$+` retains them and preserves their order.

`$*` The stem with which an implicit rule matches (Pattern Match, ,How Patterns Match).

`$(@D)`

`$(@F)` The directory part and the file-within-directory part of `$@`.

`$(*D)`

`$(*F)` The directory part and the file-within-directory part of `$*`.

`$(%D)`

`$(%F)` The directory part and the file-within-directory part of `$%`.

`$(<D)`

`$(<F)` The directory part and the file-within-directory part of `$<`.

`$(^D)`

`$(^F)` The directory part and the file-within-directory part of `$^`.

`$(+D)`

`$(+F)` The directory part and the file-within-directory part of `$+`.

`$(?D)`

`$(?F)` The directory part and the file-within-directory part of `$?`.

Zmienne specjalne używane przez GNU`make`:

`MAKEFILES`

Makefiles to be read on every invocation of `make`. MAKEFILES Variable, ,The Variable `MAKEFILES`.

`VPATH`

Directory search path for files not found in the current directory. General Search, , `VPATH` Search Path for All Dependencies.

`SHELL`

The name of the system default command interpreter, usually `/bin/sh`. You can set `SHELL` in the makefile to change the shell used to run commands. Execution, ,Command Execution.

`MAKESHELL`

On MS-DOS only, the name of the command interpreter that is to be used by `make`. This value takes precedence over the value of `SHELL`. Execution, ,MAKESHELL variable.

`MAKE`

The name with which `make` was invoked. Using this variable in commands has special meaning. MAKE Variable, ,How the `MAKE` Variable Works.

`MAKELEVEL`

The number of levels of recursion (sub-`make`s). Variables/Recursion.

`MAKEFLAGS`

The flags given to `make`. You can set this in the environment or a makefile to set flags. Options/Recursion, ,Communicating Options to a Sub-`make`.

`MAKECMDGOALS`

The targets given to `make` on the command line. Setting this variable has no effect on the operation of `make`. Goals, ,Arguments to Specify the Goals.

`CURDIR`

Set to the pathname of the current working directory (after all `-C` options are processed, if any). Setting this variable has no effect on the operation of `make`. Recursion, ,Recursive Use of `make`.

`SUFFIXES`

The default list of suffixes before `make` reads any makefiles.