10 Makefile* – rozszerzenie

Wyrażenia

Wewnątrz `$(...)` może się znajdować nie tylko pojedyńcza nazwa zmiennej, ale też bardziej skomplikowane wyrażenie w postaci wywołania jakiejś funkcji. Dla przykładu, w wyrażeniu `v=$(sort D A C B)`, `sort` to wywołanie funkcji sortującej, a pozostałe elementy, oddzielone spacją, to argumenty. Po powyższym `$v` przyjmie wartość `A B C D`.

Podstawowe funkcje na ciągach znaków to:

`$(subst from, to, text)`	Zamienia każde wystąpienie `from` przez `to` w `text`
`$(patsubst from, to, text)`	Jak wyżej, ale wewnątrz `from` może pojawić się znak `%` reprezentujący dowolny niepusty ciąg znaków. Podobny znak w `to` oznacza ciąg dopasowany do `%` we `from`.
`$(var:from=to)`	Specjalna konstrukcja równoważna `$(patsubst from, to, $(var))`. Jeżeli `from` nie zawiera znaku `%`, to jest on stawiany na początku obu argumentów, czyniąc całe wyrażenie równoważne `$(patsubst %from, %to, $(var))`
`$(strip str)`	Usuwa białe znaki z początku i końca `str`. Wielokrotne białe znaki w środku zostają zastąpione pojedyńczą spacją.
`$(filter what,str)`	poszukuje słów w `str` pasujących do któregokolwiek ze słów `what`. Np. `$(filter %.c %.o,$(pliki))` wymieni wszystkie pliki z końcówką `.o` lub `.c`.
`$(sort str)`	Sortuje słowa w `str` alfabetycznie.
`$(word n,str)`	Wybiera n-te słowo z `str`.

Język `make` dysponuje też funkcjami które operują na listach plików, np:

`$(dir nazwy)`	Dla każdej nazwy w `nazwy` wyciąga część odpowiadającą za katalog, wraz z ostatnim `/`
`$(notdir nazwy)`	Dla każdej nazwy w `nazwy` wyciąga nazwę pliku bez katalogu
`$(suffix nazwy)`	Dla każdej nazwy w `nazwy` wyciąga rozszeżenie pliku
`$(join lista1,lista2)`	Łączy parami słowa z pierwszej i drugiej listy. Dla przykładu: `$(join a b,.c .o)` stworzy `a.c b.o`
`$(wildcard glob)`	Znajduje pliki pasujące do patternu `glob`

Jeśli wartość zmiennej musi być wyliczona przez jakiś skrypt, można to zrobić za pomocą:
`VAR = $(shell <skrypt>)`
Wówczas zawartością `$(VAR)` stanie się to co skrypt wypisał na standardowe wyjście.

```
FILESREC := $(shell find . -name "*.c")
FILES := $(wildcard *.c")
```

Zapisze w zmiennej `$(FILESREC)` wszystkie pliki z końcówką `.c` w bierzącym katalogu i podkatalogach. Dla porównania – `$(FILES)` znajdzie tylko pliki w bierzącym katalogu.

Typowa konstrukcja

Przedstawiliśmy podstawowe narzędzia dostępne do konstrukcji `Makefile`. Niemniej, tak jak w każdym innym języku programowania, dany problem można rozwiązać na wiele sposób. Przyjrzyjmy się jak typowo projekt w C jest kompilowany.

W pierwszej części umieszczane są w zmiennych wszystkie programy które są potrzebne do kompilacji. W ten sposób, jeśli ktoś dostaje naszego `Makefile`-a i rzeczone programy znajdują się w inny miejscu lub pod inną nazwą, plik łatwo skorygować. We wszystkich regułach używamy tylko zmiennych a nie nazw programów zapisanych verbatim! Zasady tej nie stosuje się do uniwersalnych komend basha (np. `rm`, `cp`, `mkdir`).

```
CC=gcc
LD=gcc
```

Następnie definiowane są parametry/flagi dla tych programów. W ten sposób czytelnik może szybko zmienić parametry compilacji nie grzebiąc w komendach reguł.

```
CFLAGS=-Wall -pedantic
LDFLAGS=-fPIC
```

Niektóre flagi mogą zależeć od celu jaki budujemy. Jeśli na przykład zdefiniujemy cele `debug` i `release`, możemy tak skonfigurować flagi kompilatora:

```
debug : CFLAGS+=-g
release : CFLAGS+=-O3
release : LDFLAGS+=-O3
```

Wartości `CFLAGS` i `LDFLAGS` będą zmienione podczas wykonywania odpowiednich reguł `debug` i `release`, a także wszystkich podzadań przy realizacji ich niezbędników. Trzeba przy tym pamiętać, że reguła dla każdego celu jest uruchamiana dokładnie raz. Jeśli wykunujemy jednocześnie `debug` i `release` i oba odwołują się do tego samego podzadania, to będzie ono wykonane tylko raz, i z tylko jedną konfiguracją zmiennych.

Następnie warto w zmiennych zapisać wszystkie pliki jakie przetwarzamy: pliki `.c`, pliki `.o` itd. Na przykład można to zrobić w następujący sposób:

```
srcdir := src
builddir := build
outdir := bin
SOURCES := $(wildcard $(srcdir)/*.c)
OBJS := $(SOURCES:$(srcdir)/%.c=$(builddir)/%.o)
TARGET := $(outdir)/program
```

Gdy już wszystkie parametry kompilacji są określone, można zacząć pisać reguły. Pierwszą regułą zwykle nazywamy „`all`” – wykonuje ona wszystkie zadania z domyślnymi ustawieniami. Może być to coś prostego jak:

```
all : release
release : $(TARGET)
debug : $(TARGET)
```

W ten sposób `make`, `make all` i `make release` działają w ten sam sposób. Alternatywnie `make debug` skompiluje program w trybie debugowalnym.

Na szczęście wszystkie różnice w wywołaniu kompilatora w wersjach `release` i `debug` są już zawarte w zmiennych powyżej. Tak więc w obu przypadkach możemy użyć tych samych reguł:

```
$(TARGET) : $(OBJS)
        @mkdir -p $(dir $(TARGET))
        $(LD) $(LDFLAGS) $+ -o $@

$(builddir)/%.o : $(srcdir)/%.c
        @mkdir -p $(builddir)
        $(CC) $(CCFLAGS) $< -c -o $@
```

Na tym etapie proces kompilacji mamy już zakończony. Każdy `Makefile` powinien potrafić też posprzątać po sobie, t.j. usunąć wszystkie wygenerowane pliki. Dzieje się to standardowo za pomocą reguły `clean`.

```
clean :
        @rm -rf $(TARGET)
        @rm -rf $(OBJS)
        -@rmdir $(builddir) 2> /dev/null || true
        -@rmdir $(outdir) 2> /dev/null || true
```

Zauważmy, że nie robimy tu prostrzego `rm -rf $(builddir)$(outdir)` – w ten sposób usunęlibysmy całą zawartośc katalogów `bin` i `build` nawet jeśli użytkownik, albo jakiś inny skrypt, umieścił tam jakieś dodatkowe pliki. W wersji powyżej, usuwamy dokładnie te pliki które tworzymy. Cały katalog `bin` i `build` są usuwane tylko jeśli okazały się one na końcu puste.

Ponadto, komendy `rmdir` zostały opakowane `-@...2> /dev/null || true`. To oznacza

* `-` powoduje, że ewentualne błędy mają być zignorowane i skrypt kontynuowany.
* `2> /dev/null` przekierowuje strumień błędów do niczego.
* `|| true` powoduje, że nawet jeśli `rmdir` zwróci niezerowy kod błędu to ma on zostać zignorowany.

Innymi słowy: co by się nie działo – cicho zignoruj błędy i pracuj dalej. I w istocie: nie interesuje nas, czy udało się pomyślnie te katalogi skasować czy nie.

Na końcu jeszcze zaznaczymy które reguły nie tworzą plików. Ponadto chcemy zaznaczyć, by wybrane pliki pośrednie nie były automatycznie usuwane przez `make`.

```
.PHONY: all release debug clean
.SECONDARY: $(OBJS)
```

Generowanie listy zależności

Powyższe rozwiązanie ma jedną wadę: deklarujemy, że pliki `.o` zależą jedynie od odpowiednich plików `.c`. Przeważnie nie jest to prawdą: zależą one też (rekurencyjnie!) od wszystkich plików nagłówków jakie zostały zainkludowane w pliku źródłowym. Chcielibyśmy jakoś automatycznie otrzymać taką listę zależności dla każdego pliku `.c`

Problem ten jest na tyle popularny, że autorzy gcc dodali specjalny parametr `-M` który parsuje kod źródłowy danego pliku i generuje listę zależności:

```
DEPS := $(SOURCES:$(srcdir)/%.c=$(builddir)/%.d)

build/%.d: src/%.c
	@mkdir -p build
	@$(CC) $< -MM > $@
```

Taki plik zależności musimy jeszcze uwzględnić w samym `Makefile`-u. Można to łatwo zrobić wywołując komendę include:

```
-include $(DEPS)
```

Umieszczenie znaku - na początku powoduje, że `include` nie wypisuje błędu w przypadku gdy plik nie istnieje – a tak dzieje się zawsze, gdy dany `Makefile` uruchomiony jest po raz pierwszy – wtedy nie istnieją jeszcze pliki `.d`! Ich obecność ma sens dopiero przy drugim i kolejnym uruchomieniu.

Załóżmy, na potrzeby przykładu, że w katalogu `src` mamy plik `main.c`, który includuje 3 pliki: `<stdio.h>`, `<stdlib.h>` i `"nwd.h"`. Z pośród tych trzech, dwa należą do standardowej biblioteki języka C i zostaną pominięte (użycie komendy `-M` zamiast `-MM` by spowodowało uwzględnienie także plików standardowych). Tak więc w pliku `build/main.d` znajdziemy:

```
main.o : src/main.c src/nwd.h
```

Zauważmy, że teraz mamy dwie reguły dla pliku `.o`. Oprócz tej powyższej, także regułę ogólną `$(builddir)/%.o : $(srcdir)/%.c`. Nie stanowi to problemu: obie reguły zostaną uwzględnione.

Większym problemem jest fakt, że wygenerowany plik `.d` nie podaje ścieżki dla pliku `main.o`. Ponadto chcielibyśmy uwzględnić, że plik `.d` *też* zależy od wyliczonych plików `.c` i `.h`. Musimy zatem wyprodukować:

```
build/main.o build/main.d : src/main.c src/nwd.h
```

Na szczęście taką zmianę można łatwo wprowadzić przy pomocy `sed`-a. Korygujemy regułę do budowy plików `.d` w następujący sposób:

```
build/%.d: src/%.c
	@mkdir -p build
	@$(CC) $< -MM |\
	    sed 's=\($*\)\.o[ :]*=$(@D)/\1.o $@ : =g'\
	    > $@
```

Znak `\` na końcu linii pozwala nam na zapisanie komendy w wielu liniach i nie ma on wpływu na treść.

Podsumowanie

Podsumowując, łącząc wszystkie powyższe elementy w jedną całość otrzymujemy:

```
CC:=gcc
LD:=gcc
CFLAGS=-Wall -pedantic
LDFLAGS=-fPIC
debug : CFLAGS+=-g
release : CFLAGS+=-O3
release : LDFLAGS+=-O3
srcdir := src
builddir := build
outdir := bin
SOURCES := $(wildcard $(srcdir)/*.c)
OBJS := $(SOURCES:$(srcdir)/%.c=$(builddir)/%.o)
DEPS := $(SOURCES:$(srcdir)/%.c=$(builddir)/%.d)
TARGET := $(outdir)/program

all: release
release: $(TARGET)
debug: $(TARGET)

$(TARGET): $(OBJS)
    @mkdir -p bin
    $(LD) $+ $(LDFLAGS) -o $@
build/%.o: src/%.c build/%.d
    @mkdir -p build
    $(CC) $< $(CFLAGS) -c -o $@
build/%.d: src/%.c
    @mkdir -p build
    @$(CC) $< -MM |\
        sed 's=\($*\)\.o[ :]*=$(@D)/\1.o $@ : =g'\
        > $@

clean:
    @rm -rf $(OBJS) $(DEPS) $(TARGET)
    -@rmdir bin build 2> /dev/null || true
.PHONY: clean all release debug
.SECONDARY: $(OBJS) $(DEPS)
-include $(DEPS)
```

Drzewo typowego projektu w C/C++

Każdy programista w ciągu kariery spotyka się z większymi projektami, które składają się z wielu plików.

Mały program w C lub C++ zawiera się w pojedyńczym pliku. Taki program `program.c` kompilujemy za pomocą `gcc program.c -o program` – i gotowe!

Jednakże, większe projekty często podzielone są na większą liczbę plików. Zarządzanie i kompilacja takich projektów jest bardziej skomplikowana.

Podstawowy proces kompilacji składa się z dwóch etapów:

* Każdy plik źródłowy `.c` lub `.cpp` jest kompilowany do pliku obiektowego `.o`. Zawiera on kod maszynowy pojedyńczych funkcji oraz spis wszystkich funkcji które dany plik zawiera (i je eksportuje) lub potrzebuje (importuje). Kod maszynowy zawiera luki: adresy funkcji importowanych nie są znane.
* Wszystkie pliki obiektowe, wraz ze statycznymi bibliotekami, musza zostać połączone. Jest to zadanie linkera, który łączy wszystkie funkcje w jeden plik programu. Adresy wyeksportowane przez jeden plik są używane do uzupełnienia adresów funkcji importowanych w innych plikach.

Pojedyńcze pliki kompilujemy za pomocą:
```
gcc plik.c -c -o plik.o
```

* `<nazwa>` – argument pozycyjny to nazwa pliku który kompilujemy.
* `-c` – oznacza, że wykonujemy samą kompilację bez linkowania.
* `-o <nazwa>` – precyzuje jak ma się nazywać plik który generujemy. Domyślnie jest to nazwa pliku kompilowanego, z zastąpionym `.c` przez `.o`.

Jak już skompilujemy każdy plik z osobna, połączyć je można za pomocą *linkera* o nazwa `ld`. Łatwiej jest jednak ponownie wywołać `gcc` z plikami `.o` jako argumentami. Program `gcc` wywoła tego samego linkera, ale uzupełni argumenty o domyślne wartości dla języka C.
```
gcc plik1.o plik2.o -o program
```

* `<nazwa>` – argument pozycyjny to nazwa pliku który chcemy łączyć. Należy tu wypisać wszystkie pliki obiektowe które uprzegnio wygenerowaliśmy.
* `-o <nazwa>` – precyzuje jak ma się nazywać plik programu który generujemy. Domyślnie jest to zawsze `a.out`.

Kompilując język C++ wystarczy zamienić `gcc` przez `g++` w powyższych komendach. Oba kompilatory przyjmują te same parametry i flagi.

Zauważmy, że projekt się rozrasta: dla każdego pliku źródłowego, generujemy nowy plik

pośredni `.o`. Jest to plik tworzony podczas kompilacji, ale gdy otrzymamy główny program – taki plik nie jest już potrzebny. Z czasem może okazać się, że musimy też zawrzeć w projekcie dokumentację, lub kod źródłowy bibliotek zewnętrznych napisanych przez kogoś innego. Bardziej skomplikowane projekty moga też tworzyć jeszcze więcej różnych plików pośrednich. Dlatego każdy projekt, nawet jeśli zawiera on tylko dwa pliki źródłowe, warto podzielić na podkatalogi. Typowy podział to:

`projekt`	katalog główny. Bezpośrednio w tym katalogu umieszcza się niewiele plików. Głównie jest tam skrypt kompilujący/instalujący, krótki opis działania programu oraz informacja o prawach autorskich i licencji.
`projekt/bin`	Tu generowane są końcowe pliki binarne, takie jak programy.
`projekt/build`	Katalog na wszystkie pliki pośrednie dla procesu kompilacji.
`projekt/contrib`	Narzędzia innych osób na których dany projekt się opiera.
`projekt/data`	Dane na których program operuje
`projekt/doc`	Dokumentacja, opis programu, notatki
`projekt/include`	Jeśli projekt jest biblioteką – tu umieszane są pliki nagłówków (*header file*). Takie pliki zawierają deklaracje bez definicji.
`projekt/src`	Tu umieszczony jest kod źródłowy.
`projekt/test`	Tu jest kod testujący program, lub jego fragmenty.

Powyższa lista nie jest wyczerpująca. W danym projekcie może pojawić się więcej katalogów w zależności od potrzeb. Te są najpopularniejsze.