# Wstęp do programu *Make*

*Andrzej Görlich*

Październik 2023

Program `make` automatyzuje proces kompilacji projektu na podstawie reguł zawartych w pliku `Makefile`. Rozpoznaje, które pliki uległy zmianie (plik źródłowy jest nowszy od docelowego) i kompiluje tylko te części, które tego wymagają. Znacząco przyspiesza i upraszcza ponowną kompilację projektu. Nie jest powiązany z żadnym konkretnym językiem programowania i może być dostosowany do wielu sytuacji.

Tutaj można znaleźć dokumentację programu `make`.

> *Na końcu niniejszego dokumentu omówiono warunki jakie powinien spełniać plik* `Makefile` *w rozwiązaniach.*

## Plik Makefile

Program `make` czyta reguły zapisane przez użytkownika w pliku o nazwie `Makefile` (sugerowana), `makefile` lub `MAKEFILE`, umieszczonego w bieżącym katalogu. Sugeruję zacząć od najprostszej wersji tego pliku (obsługującego jedynie kompilację `make` i czyszczenie `make clean`) i rozbudowywać na potrzeby kolejnych zestawów.

`Makefile` składa się z **pięciu elementów**: *reguł jawnych*, *reguł wzorcowych*, *definicji zmiennych*, *dyrektyw* i *komentarzy*.

## Makefile krok po kroku

Rozważmy najprostszy przypadek, w którym plik źródłowy `hello.c` kompilujemy do tymczasowego pliku obiektowego `hello.o`, a ten do pliku wykonywalnego `hello.x` (rozszerzenie `.x` jest konwencją używaną dla plików wykonywalnych na zajęciach).

### Reguły jawne

Najprostszym typem reguł są *reguły jawne*. Reguła jawna określa kiedy i w jaki sposób należy zbudować konkretny *plik docelowy* (lub pliki) na podstawie pliku lub plików źródłowych (*zależności* lub *prerekwizyty*). Plik docelowy jest wynikiem działania reguły.

Przykład reguły jawnej:

```
hello.x: hello.o
<TAB>gcc -s -lm -o hello.x hello.o

hello.o: hello.c
<TAB>gcc -O3 -Wall -c -o hello.o hello.c
```

> ***Uwaga:*** *Symbol* `<TAB>` *oznacza znak tabulacji.*

Pierwsza linia reguły określa plik docelowy (`hello.x`), którego nazwa znajduje się przed dwukropkiem, oraz zależności (`hello.o`), czyli listy plików od których zależy cel. Modyfikacja któregokolwiek z plików źródłowych oznacza, że cel również zostanie uaktualniony. Kolejne linie określają komendy, które należy wykonać, aby zbudować plik docelowy z plików źródłowych.
Komendy **muszą** być wcięte za pomocą jednego **znaku tabulacji**, a **nie spacji**!! Proszę sprawdzić ustawienia używanego edytora tekstu.
Reguła jawna posiada następującą składnię:

```
cel: prerekwizyt ...
<TAB>komenda
<TAB>komenda
```

> ***Uwaga:*** *Kolejność reguł nie jest przypadkowa.*

Pierwsza reguła, która znajduje się w pliku `Makefile`, jest *regułą domyślną* i zostaje wykonana gdy wywołamy program `make` bez żadnych argumentów. W tym przypadku będzie to stworzenie pliku wykonywalnego `hello.x`. Jeżeli chcemy utworzyć wybrany plik docelowy, podajemy jego nazwę jako argument programu `make`:

```
$ make hello.x
```

### Reguły niejawne

*Reguła niejawna* określa kiedy i w jaki sposób otrzymać brakujący plik w oparciu o jego typ. Program *make* posiada bogaty zbiór wbudowanych reguł niejawnych. Można zdefiniować własne reguły niejawne przy użyciu *reguł wzorcowych*, które omówiono poniżej.

### Zmienne

Niektóre wyrażenia, jak na przykład opcje kompilatora, mogą się wielokrotnie powtarzać. Aby uniknąć konieczności ich modyfikacji w każdym wystąpieniu, warto użyć zmiennych. W poniższym przykładzie definiujemy zmienną z opcjami kompilatora (`CFLAGS`) i linkera (`LDFLAGS`):

```
CFLAGS  := -O3 -Wall
LDFLAGS := -s -lm

hello.x: hello.o
    $(CC) $(LDFLAGS) -o hello.x hello.o

hello.o: hello.c
    $(CC) $(CFLAGS) -c -o hello.o hello.c
```

Niektóre zmienne mają ustalone znaczenie i są predefiniowane domyślnymi wartościami (np. `CC := cc`), które można zmienić.
Lista wybranych zmiennych:

| Zmienna | Wartość domyślna | Znaczenie |
|---|---|---|
| **CC** | `cc` | Kompilator *C* |
| **CXX** | `g++` | Kompilator *C++* |
| **CPP** | `$(CC) -E` | Preprocesor *C* |
| **RM** | `rm -f` | Komenda usunięcia pliku |
| **CFLAGS** | | Flagi etapu kompilacji *C* |
| **CXXFLAGS** | | Flagi etapu kompilacji *C++* |
| **LDFLAGS** | | Flagi etapu linkowania |
| **CURDIR** | | Absolutna ścieżka do katalogu roboczego |

Ponadto importowane są także zmienne środowiskowe. Aby podstawić wartość zmiennej obejmujemy je nawiasem okrągłym (a nie klamrowym jak w *shell*u) i poprzedzamy symbolem dolara.

### Zmienne automatyczne

W poprzednim przykładzie, **cel** i **zależności** pojawiają się w pierwszej linii reguły (zawierającej dwukropek), a następnie powtarzają się w komendzie jako argumenty kompilatora. Aby uniknąć powtarzania konkretnych nazw plików docelowych i źródłowych, możemy je zastąpić zmiennymi automatycznymi, odpowiednio `$@` i `$^`:

```
CFLAGS  := -O3 -Wall
LDFLAGS := -s -lm

hello.x: hello.o
    $(CC) $(LDFLAGS) -o $@ $^

hello.o: hello.c
    $(CC) $(CFLAGS) -c -o $@ $^
```

W ramach reguł możemy stosować między innymi następujące *zmienne automatyczne*:

| Symbol | Nazwa | Opis |
|---|---|---|
| `$@` | Cel | Obiekt, który znajduje się po lewej stronie dwukropka w regule. |
| `$^` | Wszystkie prerekwizyty | Obiekty, które znajdują się po prawej stronie dwukropka. |
| `$<` | Pierwszy prerekwizyt | Obiekt, który znajduje się bezpośrednio po prawej stronie dwukropka. |
| `$?` | Wszystkie prerekwizyty nowsze niż cel | (oddzielone spacjami) |

### Reguły wzorcowe

Szybko przekonamy się, że tak zapisane reguły kompilacji (z użyciem zmiennych automatycznych), wyglądają identycznie dla wielu plików. W takim przypadku, można utworzyć regułę wzorcową:

```
CFLAGS  := -O3 -Wall -c
LDFLAGS := -s -lm

%.x: %.o
    $(CC) $(LDFLAGS) -o $@ $^

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

Nazwy plików (z wyłączeniem rozszerzenia) możemy zastąpić symbolem procent (`%`) i połączyć wiele reguł jawnych w jedną. W powyższym przykładzie mamy dwie reguły wzorcowe. Pierwsza mówi w jaki sposób *skompilować* plik źródłowy z rozszerzeniem `.c` do pliku obiektowego o tej samej nazwie bazowej ale z rozszerzeniem `.o`. Analogicznie, druga reguła określa w jaki sposób *zlinkować* plik obiektowy z rozszerzeniem `.o` do pliku wykonywalnego z rozszerzeniem `.x`.

Jeżeli chcemy wskazać, że plik docelowy zależy od niestandardowych plików źródłowych możemy utworzyć regułę bez podawania komend. Wówczas zostaną użyte komendy z odpowiedniej reguły wzorcowej:

```
hello.o: hello.c hello.h
```

W tym przypadku zmiana pliku `hello.h` wymusi rekompilację pliku `hello.o`. Ponieważ w regule wzorcowej użyto zmiennej automatycznej `$<` określającej pierwszy prerekwizyt, plik `hello.h` nie będzie argumentem kompilatora. Odwołanie do pliku nagłówkowego znajduje się w pliku źródłowym `hello.c`.

### Akcje

Akcje to reguły, których *celem* nie jest plik. Na przykład, stwórzmy akcję `clean` czyszczącą niepotrzebne pliki tymczasowe i wynikowe (`*.o` i `*.x`). Sprawdź w sekcji *zmienne* co oznacza `$(RM)`. Może okazać się, że nazwa akcji jest jednocześnie nazwą pliku w projekcie. Aby uniknąć nieporozumień należy użyć dyrektywy `.PHONY`, która mówi, że dana reguła nie jest nazwą pliku.

Program `make` wywołany bez argumentów wczytuje plik `Makefile` i wykonuje pierwszą regułę zawartą w tym pliku. Dlatego często jako pierwszy definiuje się cel `all`, który zależy od wszystkich plików wynikowych jakie chcemy stworzyć w danym projekcie. Spowoduje to ich kompilacje w razie potrzeby. Na ogół reguła dla `all` nie zawiera komend.

Przykład zawierający akcje `all` i `clean`:

```
CFLAGS  := -O3 -Wall -c
LDFLAGS := -s -lm

.PHONY: all clean

all: hello.x

hello.o: hello.c hello.h

%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $^

%.x: %.o
    $(CC) $(LDFLAGS) -o $@ $^

clean:
    $(RM) *.o *.x
```

### Komentarze

Symbol # rozpoczyna komentarz. Komentarz może rozpoczynać się od początku linii lub w jej środku. Znak # wraz z resztą linii są ignorowane. Aby użyć symbolu # należy poprzedzić go znakiem modyfikacji backslash, czyli `\#`.

### Funkcje

Program `make` posiada bogaty zbiór wbudowanych funkcji, które wykonują rozmaite operacje na argumentach. Zdefiniujmy zmienną `sources`, która przechowuje nazwy plików źródłowych:

```
sources = hello.c test.c main.c
```

Do zmiennej można się odwołać poprzez `$(sources)`. Na podstawie tej zmiennej możemy utworzyć zbiór plików obiektowych zamieniając rozszerzenie `.c` na `.o`. Użyjemy do tego funkcji `patsubst`, a wynik zapiszemy pod zmienną `objects`:

```
objects = $(patsubst %.c,%.o,$(sources))
```

Składnia funkcji `patsubst` (*pattern substitution*) wygląda następująco: `$(patsubst wzor_wejsciowy, wzor_wynikowy, lista_wyrazow)`. Funkcja ta każdy wyraz z listy wyrazów kojarzy ze wzorem wejściowym i zamienia go tak, by pasował do wzorca wyjściowego, pozostawiając niezmienione części, które zostały schowane za znakiem `%`. Podobnie funkcja `addsuffix` dodaje rozszerzenie do nazw plików według następującej składni: `$(addsuffix suffix,names...)`.
Szczegółową listę funkcji wbudowanych znajdziemy tutaj.

> *Zamiast wieloznaczników powłoki (`*.o *.x`) w regule* `clean`*, lepiej użyć funkcji* `wildcard`*,* `$(RM) $(wildcard *.[ox])`*. Różnica stanie się widoczna w przypadku, gdy nie istnieje żaden plik o rozszerzeniu* `.o` *lub* `.x`*.*

### Opcje programu `make`

Przydatne opcje programu `make`: `-B`, `-n`, `-p` (więcej informacji `make man`).
Program `make` automatycznie usuwa pliki pośrednie (np. `*.o`).

Opcja `-MM` kompilatora `gcc` pozwala uzyskać informację o zależnościach, za pomocą których można stworzyć prosty `Makefile`:

```
gcc -MM module.c
```

## Przykład podstawowego pliku `Makefile`

Rozważmy projekt w *C*, który składa się z pięciu plików źródłowych,
`main.c, test.c, test.h, module.c, module.h`
Służą one do stworzenia docelowego pliku wykonywalnego `program.x` na podstawie reguł zawartych w poniższym przykładowym pliku `Makefile`:

```
CC      := gcc
CFLAGS  := -Wall
LDFLAGS := -s
LDLIBS  := -lm

sources = module test main
objects = $(addsuffix .o,$(objects))    # Przykład użycia funkcji addsuffix

.PHONY: clean

module.o: module.c module.h             # Definicja zależności
test.o: test.c test.h                   # Plik .c można pominąć,
main.o: main.c test.h                   # zostanie dodany przez regułę wzorcową

program.x: $(addsuffix .o,$(sources))

%.o: %.c                                # Reguła wzorcowa .c [+ .h] -> .o
    $(CC) $(CFLAGS) -c -o $@ $<

%.x: %.o                                # Reguła wzorcowa .o + ... -> .x
    $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)

clean:                                  # Akcja
    $(RM) program.x $(objects)          # Lub $(wildcard *.[ox])
```

### Zadanie

Dopisać do `Makefile` akcje: `tar` (archiwizacja bieżącego katalogu do `.tar.gz`), `all` (domyślna - powinna zostać umieszczona jako pierwsza - reguła, która kompiluje wszystkie programy w zestawie).

## I Plik `Makefile` w rozwiązaniach

Należy napisać plik `Makefile` w taki sposób, aby komenda `make` kompilowała **wszystkie** programy w zestawie. Programy, które Państwo piszą muszą się kompilować z opcją `-Wall` (proszę dodać `-Wall` do zmiennej `CFLAGS` w `Makefile`) bez żadnych błędów, ani ostrzeżeń. Ostrzeżenia zazwyczaj świadczą o niepoprawnym kodzie (jak również pewnym dyletanctwie).

### Uwagi

- Zdefiniować i używać zmienne, np. `CFLAGS`.
- Używać zmiennych automatycznych (`$@`, `$^`, `$<`).
- Zdefiniować i użyć reguły wzorcowe.
- Etap linkowania (`.o → .x`) powinien używać flag w `LDFLAGS`, a nie `CFLAGS`!
- Nie dodawać komend, jeżeli są one już w regułach wzorcowych.
- Korzystać z reguł wbudowanych.
- Pamiętać o zależności od plików nagłówkowych.
- Wbudowana zmienna `RM` już zawiera flagę `-f`. Nie używać flagi `-r`.
- Nie dodawać akcji `run`.

Fragment

```
main.x: main.o procinfo.o
    gcc -s -lm -o main.x main.o procinfo.o

main.o: main.c procinfo.h
    gcc -Wall -O3 main.c
```

powinien wyglądać następująco

```
main.x: main.o procinfo.o
main.o: main.c procinfo.h

%.x: %.o
    $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)
```