Języki Programowania, Makefile

Do tej pory korzystaliśmy z bardzo prostego pliku **Makefile**, który miał jedną komendę, której wykonanie uruchamiało nam kompilator g++ (zamiast wpisywanie tego „z palca” w linii poleceń), czyli np.:

```
all:
	g++ -Wall Klasa.cpp main.cpp -o main
```

W ogólności narzędzie `make` ma dużo więcej możliwości i jest wykorzystywane w dużych projektach informatycznych składających się z ogromnej liczby plików źródłowych. Zrobimy więc dzisiaj bardziej rozbudowany **Makefile**.
Struktura plików **Makefile** składa się z tak zwanych reguł i jest następująca:

```
CEL: SKŁADNIKI
	KOMENDA
```

gdzie `CEL` jest nazwą pliku wynikowego, który jest tworzony z plików wymienionych jako `SKŁADNIKI`, zaś `KOMENDA` jest komendą (przed nią musi stać tabulacja), która tworzy `CEL` z plików w `SKŁADNIKI`.
Przykład takiej reguły:

```
main: main.cpp klasa.cpp
	g++ -Wall klasa.cpp main.cpp -o main
```

Pole SKŁADNIKI nie jest wymagane, jeśli pliki składowe wymienione są w komendzie explicite (w tym przypadku nie muszą zatem tam być podane – podobnie jak w naszym prostym Makefile'u).
W dużych projektach tworzymy nasz ostateczny wynikowy plik tworzony jest z wielu reguł pomocnicznych. Na przykład:

```
main: klasa.o
	g++ -Wall klasa.o main.cpp -o main
klasa.o: klasa.cpp
	g++ -Wall klasa.cpp -c -o klasa.o
```

Załóżmy, że mamy klasę składającą się z plików **klasa.cpp** oraz **klasa.h**. Druga komenda tworzy nam plik **klasa.o**, który jest tzw. „object file” - jest to skompilowany fragment kodu (nie program), zawierający wszystkie zmienne i metody klasy. Wiele „object files” można łączyć w biblioteki. Ponadto tego typu organizacja projektu pozwala na kompilowanie wybranych fragmentów i ewentualne poprawianie błędów tylko w danej części bez kompilowania całego programu.
W pliku **Makefile** możemy też używać zmiennych, podobnie jak w skryptach bash'owych. Na przykład (dla przykładu tym razem z dwoma klasami):

```
OBJS=klasa1.o klasa2.o

main: $(OBJS)
	g++ -Wall $(OBJS) main.cpp -o main
klasa1.o: klasa1.cpp
	g++ -Wall klasa1.cpp -c -o klasa1.o
klasa2.o: klasa2.cpp
	g++ -Wall klasa2.cpp -c -o klasa2.o
```

Używamy również tak zwanych zmiennych standardowych, które ułatwiają nam modyfikację parametrów zarządzających kompilacją projektu. W naszym przypadku możemy więc stworzyć takie zmienne (tak nazywa się zwykle zmienne w plikach **Makefile**):
`CXX` – kompilator C++
`CXXFLAGS` – flagi kompilatora C++ (np. `-Wall`)
`LFLAGS` – flagi linkera (gdy dołączamy jakąś zewnętrzną bibliotekę, jeszcze tego nie robiliśmy, więc zostawiamy tą zmienną pustą)

Możemy więc teraz napisać nasz **Makefile** jako:

```
CXX=g++
CXXFLAGS=-Wall
LFLAGS=

OBJS=klasa1.o klasa2.o

main: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) $(OBJS) main.cpp -o main
klasa1.o: klasa1.cpp
     $(CXX) $(CXXFLAGS) klasa1.cpp -c -o klasa1.o
klasa2.o: klasa2.cpp
     $(CXX) $(CXXFLAGS) klasa2.cpp -c -o klasa2.o
```

Oprócz tego, możemy jeszcze korzystać z tak zwanych zmiennych automatycznych. Zmienne automatyczne to zmienne dynamiczne, które zmieniają się w zależności od przetwarzanego pliku:
< - aktualnie przetwarzany plik z listy składników
@ - nazwa pliku docelowego
^ - składniki (wkleja w to miejsce to co znajduje się w polu SKŁADNIKI)
Przy użyciu zmiennych automatycznych nasz **Makefile** będzie zatem miał postać:

```
CXX=g++
CXXFLAGS=-Wall
LFLAGS=

OBJS=klasa1.o klasa2.o

main: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main.cpp $^ -o $@
klasa1.o: klasa1.cpp
     $(CXX) $(CXXFLAGS) $< -c -o $@
klasa2.o: klasa2.cpp
     $(CXX) $(CXXFLAGS) $< -c -o $@
```

Widzimy, że pliki z pola SKŁADNIKI wybierane są teraz automatycznie przez zmienne automatyczne. Kolejnym uproszczeniem jest wykorzystanie wzorca reguły. Pozwoli to na skrócenie pliku **Makefile**. Widzimy w powyższym przykładzie, że musimy explicite stworzyć pliki .o dla obu klas. Można sobie wyobrazić, że w przypadku wielkich projektów takich plików (zatem i linijek w pliku **Makefile**) byłoby tysiące. Z wykorzystaniem wzorca nasz **Makefile** skracamy do postaci:

```
CXX=g++
CXXFLAGS=-Wall
LFLAGS=

OBJS=klasa1.o klasa2.o

main: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main.cpp $^ -o $@
$(OBJS): %.o: %.cpp
     $(CXX) $(CXXFLAGS) $< -c -o $@
```

Co więcej, reguła tworząca pliki .o jest regułą domyślną i nie trzeba jej explicite pisać. Poniższy

Makefile zrobi więc dokładnie to samo (stworzy pliki .o i z nich program wynikowy):

```
CXX=g++
CXXFLAGS=-Wall
```

```
LFLAGS=

OBJS=klasa1.o klasa2.o

main: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main.cpp $^ -o $@
```

Oprócz reguł do tworzenia możemy stworzyć regułę do usuwania skompilowanego programu, regułę `clean`.

```
CXX=g++
CXXFLAGS=-Wall
LFLAGS=

OBJS=klasa1.o klasa2.o

main: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main.cpp $^ -o $@
clean:
     rm -rf *.o main
```

Jak widzimy, reguła ta usuwa nam wszystkie pliki powstałe przy kompilacji (pliki .o oraz plik z programem). Co jednak zrobić, gdy mamy 2 programy wynikowe i chcemy używać jednego pliku **Makefile**? Nic trudnego, piszemy dwie reguły i możemy też dopisać regułę `all`, która od razu skompiluje nam oba, np.:

```
CXX=g++
CXXFLAGS=-Wall
LFLAGS=

OBJS=klasa1.o klasa2.o

all: main1 main2
main1: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main1.cpp $^ -o $@
main2: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main2.cpp $^ -o $@
clean:
     rm -rf *.o main
```

Reguły `clean` i `all` są regułami specjalnymi w tym sensie, że `clean` i `all` nie są nazwami plików. Gdyby jednak zdarzyło się, że w katalogu bieżącym istnieje plik o nazwie `all` lub `clean` to reguły te mogłyby nie działać! Aby temu zapobiec trzeba by poinstruować make'a, że `all` i `clean` nie są nazwami plików. Do tego celu służy specjalna reguła o nazwie `.PHONY.`:

```
CXX=g++
CXXFLAGS=-Wall
LFLAGS=

OBJS=klasa1.o klasa2.o

all: main1 main2
main1: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main1.cpp $^ -o $@

main2: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main2.cpp $^ -o $@
clean:
```

```
	rm -rf *.o main
.PHONY: all clean
```

Tak przygotowany **Makefile** możemy następnie łatwo modyfikować i dostosowywać w celu zarządzania naszą kompilacją (zmieniając flagi kompilatora, pliki składowe projektu itp.). Z reguły więc, w przypadku niedużych projektów, mamy szablon takiego Makefile'a, który tylko dostosowujemy.