Laboratorium 2:
„Biblioteki statyczne w języku C”

mgr inż. Arkadiusz Chrobot

9 października 2008

1 Wprowadzenie

Pisząc programy w języku C, szczególnie te duże, nie musimy umieszczać całości kodu źródłowego w jednym pliku. Podobnie jak w innych językach programowania możemy podzielić go na niezależne części, które mogą być kompilowane osobno i w razie konieczności włączane do innych programów. Takie pliki nazywamy bibliotekami. W języku C z bibliotekami związanie są *pliki nagłówkowe*, które ułatwiają korzystanie z nich. Ta instrukcja wyjaśnia w jaki sposób stworzyć własną bibliotekę statyczną i jak jej użyć w innym programie.

2 Definicje i deklaracje zmiennych oraz funkcji

Aby móc użyć w programie zmiennej zdefiniowanej w innym pliku musimy użyć jej deklaracji. *Deklaracja zmiennej* informuje kompilator, że zmienna o podanej nazwie i typie została już wcześniej w kodzie zdefiniowana. *Deklaracja* zmiennej informuje kompilator, że w miejscu jej umieszczenia trzeba przydzielić pamięć na tę zmienną i ją zainicjalizować. Każdy z plików z kodem źródłowym może zawierać osobną deklarację jednej i tej samej zmiennej — nawet ten, który zawiera jej definicję (przydaje się, jeśli chcemy użyć zmiennej przed jej zdefiniowaniem). Definicja zmiennej może występować tylko w jednym pliku. Aby zadeklarować zmienną należy przed jej typem i nazwą umieścić słowo kluczowe **extern**. Definicja polega na podaniu typu i nazwy zmiennej. Jeśli w żadnym z plików nie zadeklarujemy zmiennej, to jej definicja stanie się równocześnie jej deklaracją. Dotychczas w programach używaliśmy właśnie takiej łącznej deklaracji i definicji. Odwrotna sytuacja, czyli podanie deklaracji zmiennej bez definicji powoduje błąd czasu kompilacji. Poniżej znajduje się przykładowy kod źródłowy, który został podzielony na dwa pliki.

external.c

```
int foo;
```

prog1.c

```
#include<stdio.h>

extern int foo;

int main(void)
{
        printf("%d\n",foo);
        return 0;
}
```

Czytając kod można zauważyć, że w pliku o nazwie „prog1.c" została użyta zmienna „foo", która jest w tym pliku jest jedynie zadeklarowana. Jej definicja znajduje się w pliku „external.c". Wartość początkowa tej zmiennej wynosi „0", bo jest to zmienna globalna. Jeśli chcielibyśmy w pliku „external.c" umieścić dodatkową zmienną, widoczną tylko w tym pliku to przed jej definicją powinniśmy umieścić słowo kluczowe **static**.

Podobnie jak ze zmiennymi możemy postąpić z funkcjami. Aby użyć funkcji zdefiniowanej w innym pliku wystarczy umieścić jej nagłówek przed miejscem jej

wywołania, poprzedzając go słowem kluczowym **extern**. W nagłówku tym można pominąć nazwy argumentów, zostawiając wyłącznie typy. Oto przykład:

external2.c

```
#include<stdio.h>

void print(int variable)
{
        printf("%d\n",variable);
}
```

prog2.c

```
extern int foo;
extern void print(int);

int main(void)
{
        print(foo);
        return 0;
}
```

Program zapisany w pliku „prog2.c" korzysta również ze zmiennej „foo" z pliku „external.c" Proszę zwrócić uwagę, że tylko w pliku „external2.c" konieczne było włączenie pliku nagłówkowego „stdio.h".

Kompilacja tak napisanych programów składa się z kilku etapów. Najpierw musimy skompilować pliki, w których zdefinowane są funkcje i/lub zmienne. Tę kompilację należy wykonać tak jak pokazuje przykład. Opcja *-c* powoduje, że kompilator

```
gcc -Wall -c library.c
```

utworzy plik obiektowy [1], o takiej samej nazwie jak plik z kodem źródłowym, ale o rozszerzeniu „.o", który będzie można połączyć (ang. *link*) z innym plikiem. Po wykonaniu kompilacji wszystkich bibliotek możemy przystąpić do skompilowania programu głównego. Kompilację tą wykonuje się podobnie jak w kolejnym zamieszczonym przykładzie.

```
gcc -Wall -o program program.c library.o library2.o
```

3 Pliki nagłówkowe

Dzięki plikom nagłówkowym możemy zebrać w jednym miejscu deklaracje wszystkich zmiennych i funkcji, których definicję są rozproszone po wielu plikach. Włączając odpowiednio przygotowany plik nagłówkowy do innych plików z kodem źródłowym unikamy konieczności wielokrotnego przepisywania deklaracji wspomnianych elementów programu. Spróbujmy zastosować plik nagłówkowy w ostatnim przykładowym programie z poprzedniego rozdziału.

[1] Ta nazwa nie ma nic wspólnego z programowaniem zorientowanym obiektowo.

header.h

```
#ifndef HEADER_H
#define HEADER_H

extern int foo;
extern void print(int);

#endif
```

external.c

```
#include"header.h"
int foo;
```

external2.c

```
#include<stdio.h>
#include"header.h"

void print(int variable)
{
        printf("%d\n",variable);
}
```

prog2.c

```
#include"header.h"

int main(void)
{
        print(foo);
        return 0;
}
```

Jak widać na załączonych listingach deklaracje zmiennej „foo" oraz funkcji „print" zostały umieszczone w pliku nagłówkowym. Można również zauważyć, że te deklaracje otoczone są instrukcjami poprzedzonymi znakiem „#". Są to instrukcje preprocesora, które zapobiegają wielokrotnemu włączeniu wspomnianych deklaracji w wypadku gdybyśmy mieli do czynienia z kilkoma plikami nagłówkowymi, które wzajemnie się włączają. Zasada działania tego tych instrukcji zwanych *wartownikami pliku nagłówkowego* zostanie wyjaśniona w następnej instrukcji. Zaważmy, że stworzony przez nas plik nagłówkowy został włączony do wszystkich plików z rozszerzeniem „.c" za pomocą instrukcji *include*, ale nazwa tego pliku nie jest ujęta w nawiasy trójkątne, tylko w cudzysłów. Informuje to kompilator, że powinien poszukać naszego pliku nagłówkowego w katalogu bieżącym. Moglibyśmy umieścić nazwę naszego pliku w nawiasach trójkątnych, ale wówczas musielibyśmy dołączyć katalog z naszym plikiem nagłówkowym do listy katalogów, które będą przeszukiwane przez kompilator (opcja *-I*). Włączenie pliku nagłówkowego do pliku z kodem źródłowym, gdzie definiowana jest zmienna lub funkcja w opisywanym przykładzie

jest opcjonalne, ale pozwala kompilatorowi na etapie kompilacji sprawdzić, czy nie ma rozbieżności między definicją i deklaracją funkcji i zmiennej. W sytuacji gdy plik nagłówkowy zawiera definicje typów lub makr preprocesora takie włączenie jest konieczne. Reasumując - zawsze opłaca się włączyć plik nagłówkowy do wszystkich plików źródłowych z nim związanych. W pliku nagłówkowym powinniśmy unikać umieszczania definicji zmiennych i funkcji, poprzestając tylko na deklaracjach. Słowo kluczowe **extern** przed nagłówkiem funkcji jest opcjonalne.

4 Program *make*

Kiedy mamy kod źródłowy programu podzielony na kilka plików (jednostek kompilacji) konieczne staje się kilkukrotne wywołanie kompilatora z linii poleceń. Taka metoda kompilacji jest w przypadku dużych projektów zbyt czasochłonna. Rozwiązaniem tego problemu jest użycie narzędzia do kompilacji rozłącznej, jakim jest program *make*. Wielokrotna kompilacja przy pomocy tego programu sprawdza się wyłącznie do wypisania jego nazwy w linii poleceń i naciśnięcia klawisza „Enter". Zanim jednak tak się stanie należy przygotować specjalny plik o nazwie „Makefile", który będzie zawierał odpowiednie reguły dla takiej kompilacji. Oto plik „Makefile" dla przykładu, który był omawiany w poprzednim rozdziale.

Makefile

```
CC = gcc
OFLAGS = -Wall -o
CFLAGS = -Wall -c

prog2: prog2.c external2.o external.o
	$(CC) $(OFLAGS) prog2 prog2.c external.o external2.o

external2.o: external2.c
	$(CC) $(CFLAGS) external2.c

external.o: external.c
	$(CC) $(CFLAGS) external.c

clean:
	rm -f *~
	rm -f *.o
	rm -f *.a

cleanall: clean
	rm -f prog2
```

Na początku tego pliku znajdują się deklaracje zmiennych, które posłużą do zdefiniowania odpowiednich reguł. Zmiene te definowane są poprzez podanie ich nazwy i nadanie im wartości za pomocą operatora „=". W powyższym przykładzie tymi wartościami są nazwa kompilatora oraz flagi kompilacji. Pierwsza reguła która zostanie zdefiniowana w pliku „Makefile" jest wykonywana domyślnie. W przykładowym pliku ta reguła nazywa się *prog2* i powoduje skompilowanie pliku zawierającego funkcję *main*. Regułę definiuje się podając jej nazwę, która może (lecz nie musi)

być taka sama jak nazwa pliku powstającego w wyniku wykonania tej reguły. Po nazwie stawiamy dwukropek. Po nim należy podać nazwy reguł, które muszą być wykonane przed definiowaną regułą. Opcjonalnie można również podać nazwy plików z kodem źródłowym niezbędnym do wykonania reguły. Działania wykonywane w ramach reguły definiowne są w następnych wierszach. W omawianym przykładzie jest to po prostu takie samo wywołanie kompilatora jakie do tej pory ręcznie wpisywaliśmy w powłoce systemowej. Jedyna różnica polega na zastąpieniu nazwy kompilatora i flag wartościami odpowiednich zmiennych. Wartości te uzyskujemy umieszczając nazwę zmiennej w nawiasach okrągłych, a przed całością stawiając znak dolara. Definicję poszczególnych reguł oddzielamy pustym wierszem. Po zdefiniowaniu pliku „Makefile" w katalogu z kodem źródłowym programu wystarczy wydać w powłoce systemowej polecenie „make" żeby wykonać kompilację programu. Tak jak napisano to wcześniej zostanie wykonana reguła *prog2* oraz wszystkie inne od których ona zależy. Nie zostaną wykonane natomiast reguły *clean* i *cleanall*. Pierwsza z tych reguł jest odpowiedzialna za usunięcie zbędnych plików powstałych podczas kompilacji i edycji plików, a ostania za usunięcie wszystkich plików powstałych podczas kompilacji. Aby je wykonać należy w powłoce systemowej po słowie „make" wpisać nazwę reguły, która ma być wykonana. Tak możemy postąpić z dowolną regułą zdefiniowaną w pliku „Makefile". Możemy również stworzyć regułę, która będzie kompilowała więcej niż jeden program. Wystarczy dla każdego z tych programów stworzyć osobną regułę kompilacji, a następnie jako pierwszą w pliku umieścić regułę która na liście zależności będzie miała wymienione te reguły. Reguła ta może być pusta, tzn. po pierwszym wierszu będą dwa wiersze puste. Program „make" porównuje czasy powstania plików wynikowych z czasami modyfikacji plików źródłowych. Jeśli plik wynikowy jest „starszy" od źródłowego, to nie wykonuje związanej z nim reguły kompilacji. Jeśli jest odwrotnie lub jeśli plik wynikowy nie istnieje to reguła jest wykonywana. Jeśli z jakiś powodów nie możemy użyć nazwy „Makefile" to możemy użyć nazwy „makefile" lub dowolnej innej. W tym ostatnim przypadku należy programowi „make" wskazać nazwę pliku z regułami, dodając do jego wywołania przełącznik *-f* i podając nazwę tego pliku. Program „make" jest dosyć dobrze zintegrowany z edytorem *Vim*. Możemy wywołać program „make" z poziomu tego edytora. Wystarczy w trybie poleceń wydać polecenie *:cmake*. Jeśli podczas kompilacji pojawią się błędy to możemy przestawiać kursor do wiersza w którym występują za pomocą poleceń *:cnext* i *:cprev*. Programu „make" możemy używać również dla programów napisanych w innych językach programowania.

5 Biblioteki statyczne i dynamiczne

Jak zostało to napisane na wstępie ta instrukcja dotyczy wyłącznie bibliotek łączonych statycznie z plikiem wynikowym. Takie biblioteki są w całości włączane do kodu wynikowego kompilowanych z nimi programów, zwiększając tym samym objętość plików wynikowych. Istnieją również biblioteki dynamiczne, które nie mają tej wady. Proces ich tworzenia jest prawie taki sam jak bibliotek statycznych, różni je sposób kompilacji. Ten temat nie będzie tu szerzej opisywany. Biblioteki statyczne można archiwizować programem *ar*. Pliki „.o" są pakowane do plików „.a" i zastępują je podczas kompilacji. Oto przykład użycia programu „ar": *ar rs archiwum.a biblioteka.o*.