Instrukcja do zajęć laboratoryjnych

Język ANSI C (w systemie LINUX)

wersja: 1.22

Nr ćwiczenia:	2	
Temat:	**Tworzenie najprostszych programów w języku C w środowisku LINUX**	
Cel ćwiczenia:	Celem ćwiczenia jest zapoznanie studenta z zasadami pracy z kompilatorem GNU cc (gcc) dostępnym we wszystkich dystrybucjach systemu LINUX.	
Wymagane przygotowanie teoretyczne:	Informacje podane na wykładzie.	
Sposób zaliczenia:	Sprawozdanie w formie pisemnej.	[ ]
	Pozytywna ocena ćwiczenia przez prowadzącego pod koniec zajęć.	[x]

1. Konwencje przyjęte w instrukcji

`Czcionka o stałej szerokości`

Nazwy programów, poleceń, katalogów, wyniki działania wydawanych poleceń.

`Czcionka o stałej szerokości pogrubiona`

Podaje tekst, który należy dosłownie przepisać. W przypadku plików źródłowych wyróżnia istotniejsze fragmenty.

`Czcionka o stałej szerokości kursywą`

Tekst komentarza w przykładowych sesjach przy terminalu.

`Czcionka o stałej szerokości kursywą pogrubiona`

Wyróżnia istotniejsze fragmenty wyników działania wydawanych poleceń.

2. Uwagi wstępne

Celem ćwiczenia jest zapoznanie studenta z podstawami pracy z kompilatorem GNU cc (`gcc`), który *de facto* jest standardem we wszystkich dystrybucjach systemu LINUX. Praca z nim nie jest specjalnie trudna (przynajmniej do czasu, gdy nie musimy używać jego bardzo zaawansowanych możliwości), jednak wymaga poznania paru podstawowych jego elementów.

Przede wszystkim należy uświadomić sobie (i to zaakceptować), że praca z tym kompilatorem odbywać będzie się wyłącznie w trybie tekstowym. Innymi słowy wszystkie polecenia będą wydawane wprost z konsoli systemowej. Ponieważ zwykle ilość tych poleceń jest dość duża, niemal zawsze będziemy posługiwać się programem `make` ułatwiającym tworzenie bardziej złożonych aplikacji.

Praca z programem `make` będzie tematem jednego z kolejnych laboratoriów. Na początek możemy spokojnie obejść się bez tego narzędzia.

3. Kompilacja najprostszego programu

Najprostszy program w języku C może mieć następującą „klasyczną" postać (uwaga: numery poszczególnych linii nie są częścią kodu źródłowego. Umieszczono je wyłącznie w celu łatwiejszego odwoływania się do niego):

```
1 #include<stdio.h>
2 int main(void)
3 {
4 printf("Hello, world !\n");
5 return(0);
6 }
```

Zadaniem powyższego programu jest, jak pewnie nie trudno się domyśleć, wyświetlenie napisu „`Hello, world !`" na ekranie komputera.

W linii 1 umieszczono tzw. *dyrektywę* `#include`, która zleca dołączyć w czasie kompilacji programu zawartość tzw. *pliku nagłówkowego* o nazwie `stdio.h` (tam zawarta jest min. definicja funkcji standardowej `printf` i wiele, wiele innych informacji. Warto choćby pobieżnie zapoznać się z jego zawartością, choć prawdopodobnie na początku większość jego zawartości będzie dla nas kompletnie nieczytelna).

W linii 2 umieszczona jest deklaracja funkcji `main`. W każdym pliku źródłowym w języku C musi istnieć przynajmniej jedna funkcja i musi ona nazywać się `main`. Napis `int` oznacza, że funkcja zwracać będzie liczbę (zmienną) typu całkowitego, natomiast napis `void` oznacza, że funkcja nie otrzymuje z zewnątrz żadnych parametrów.

W liniach 3-6 znajduje się ciało funkcji `main`. Wewnątrz niej znajduje się standardowa funkcja o nazwie `printf`, której zadaniem jest wyświetlenie na ekranie podanego jako jej argument napisu. Znaki sterujące `\n` nakazują po wyświetleniu napisu ustawić kursor w nowej linii. Po wyświetleniu napisu funkcja kończy działanie i zwraca do środowiska wywołującego wartość `0`, co zwykło się interpretować jako poprawne zakończenie działania programu.

Aby skompilować powyższy program wydajemy polecenie:

```
$ gcc hello.c
```

W wyniku otrzymujemy skompilowany plik o nazwie `a.out`. Gdy chcemy, aby plik wynikowy miał inną nazwę (a zwykle tak chcemy), wówczas wydajemy polecenie kompilacji w takiej jak poniżej postaci:

```
$ gcc hello.c -o hello
```

Wówczas powstanie plik wynikowy o nazwie `hello` (bez typowego dla systemów DOS / Windows rozszerzenia `.exe` – choć nic oczywiście nie stoi na przeszkodzie, aby takie rozszerzenie nadać plikowi wynikowemu. W środowiskach UNIX nie jest to jednak praktykowane).

Uwaga: Ostatnia linia w pliku źródłowym MUSI być poprawnie zakończona, tzn. ostatnim znakiem musi być znak końca linii (LF – ang. *line feed*, kod ASCII `0A`). Gdy będzie inaczej kompilator wygeneruje ostrzeżenie:

```
$ gcc hello.c -o hello
hello.c:6:2: warning: no newline at end of file
```

Poprawny plik źródłowy „Hello world" (oglądany jako kody szesnastkowe) będzie wyglądał więc jak poniżej. Znak końca linii, o którym mowa wyżej, został powiększony i wytłuszczony:

```
2369 6E63 6C75 6465 3C73 7464 696F 2E68   #include<stdio.h
3E0A 696E 7420 6D61 696E 2876 6F69 6429   >.int main(void)
0A7B 0A70 7269 6E74 6628 2248 656C 6C6F   .{.printf("Hello
2C20 776F 726C 6420 215C 6E22 293B 0A72   , world !\n");.r
6574 7572 6E28 3029 3B0A 7D0A             eturn(0);.}.
```

O tym, że plik powstał możemy przekonać się wydając polecenie `ls -l` systemu operacyjnego. Zwróćmy uwagę na atrybut „`x`" tego pliku. Oznacza on, że plik jest wykonywalny (ang. *executable*). Aby uruchomić skompilowany właśnie program wydajemy następujące polecenie:

```
$ ./hello
Hello, world !
```

(uwaga: kropka oznacza, że chcemy wykonać program znajdujący się w katalogu bieżącym. Bez tej kropki system, operacyjny będzie poszukiwał wskazanego pliku w katalogach zapisanych w zmiennej systemowej `$PATH` a to w niektórych sytuacjach może być bardzo groźne. Czy domyślasz się dlaczego?).

Kompilując program kompilatorem `gcc` możemy podać kompilatorowi różne szczegółowe informacje w jaki sposób ma kompilować nasze programy. Informacje te podajemy w formie tzw. przełączników kompilatora. Ich pełen opis zajmuje kilkanaście stron w dokumentacji (patrz polecenie `man gcc` lub `http://gcc.gnu.org`). Poniżej podajemy tylko kilka przykładowych opcji:

`-Ikatalogu`	wstawia `katalog` na początek listy katalogów, które kompilator będzie przeglądał w poszukiwaniu plików nagłówkowych
`-Lkatalogu`	wstawia `katalog` na początek listy katalogów, które kompilator będzie przeglądał w poszukiwaniu bibliotek
`-lcostam`	dołącza bibliotekę o nazwie `libcostam`. Uwaga: w systemach z rodziny UNIX przyjęło się, że biblioteki mają nazwy rozpoczynające się od słowa `lib`. Wobec tego, gdy napiszemy `-lcostam` kompilator `gcc` będzie szukał na dysku pliku o nazwie `libcostam.a` lub `libcostam.so`. Rozszerzenia `.a` oraz `.so` są standardowymi rozszerzeniami bibliotek w systemie UNIX i używając przełącznika `-l` nie podajemy ich jawnie
`-Wall`	wyświetla wszystkie ostrzeżenia kompilatora `gcc`
`-w`	wyłącza wszystkie ostrzeżenia kompilatora. Używanie tej opcji nie jest zalecane.
`-pedantic`	Wyświetla wszystkie ostrzeżenia wymagane przez ANSI C
`-pedantic-errors`	Wyświetla wszystkie błędy wymagane przez ANSI C
`-werror`	ostrzeżenia zamienia na błędy, zatrzymując kompilację

`-g`	wstawia do pliku wynikowego (binarnego) standardowe informacje wyszukiwania błędów (tzw. informacje dla debuggera – będzie o nim mowa w jednym z kolejnych ćwiczeń)
`-ggdb`	podobnie jak `-g`, ale informacji tych jest znacznie więcej
`-ansi`	ciekawy przełącznik o walorach „dydaktycznych". Ścisła kontrola zgodności ze standardem ANSI C.

Poniżej wykonano pewien ciąg czynności demonstrujących działanie tych przełączników. Aby „zmusić" kompilator go wygenerowania ostrzeżenia zmieniono druga linię w przykładowym programie na:

```
void main(void)
```

Kompilator będzie wówczas ostrzegał nas o niepoprawnej definicji funkcji `main`. Definicja mówi, że funkcja nie zwraca żadnej wartości (słowo kluczowe `void`), choć w rzeczywistości funkcja zwraca wartość typu całkowitego.

Zwykle powinniśmy dążyć do tego, aby eliminować przyczyny pojawiania się ostrzeżeń kompilatora. W bardzo prostych programach może nie jest to aż tak istotne, jednak w złożonych projektach ostrzeżenia kompilatora mogą być zwiastunem trudnych do zdiagnozowania błędów w działaniu programów.

Zwróćmy również uwagę na zmianę wielkości pliku wynikowego po użyciu opcji `-g` oraz `-ggdb`.

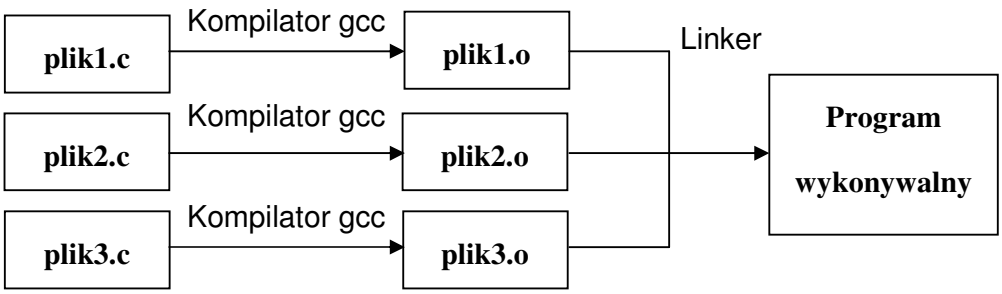
```
Na początku mamy tylko plik źródłowy.

$ ls -l
-rw-r--r--    1 root     root           82 Sep 21 21:53 hello.c

Kompilujemy plik źródłowy. Powstaje plik wynikowy a.out. Jest to plik
wykonywalny (atrybut x).

$ gcc hello.c
$ ls -l
-rwxr-xr-x    1 root     root         4819 Sep 22 00:12 a.out
-rw-r--r--    1 root     root           82 Sep 21 21:53 hello.c

Wykonujemy program. Uwaga na kropkę.

$ ./a.out
Hello, world !

Kompilujemy plik źródłowy. Powstaje plik wynikowy hello.

$ gcc hello.c -o hello
$ ls -l
-rwxr-xr-x    1 root     root         4819 Sep 22 00:13 a.out
-rwxr-xr-x    1 root     root         4819 Sep 22 00:13 hello
-rw-r--r--    1 root     root           82 Sep 21 21:53 hello.c

Kompilujemy plik źródłowy. Dodajemy informacje dla debuggera. Rozmiar
pliku wynikowego powiększa się.

$ gcc hello.c -o hello -g
$ ls -l
-rwxr-xr-x    1 root     root         4819 Sep 22 00:13 a.out
-rwxr-xr-x    1 root     root        14063 Sep 22 00:14 hello
```

```
-rw-r--r--    1 root     root           82 Sep 21 21:53 hello.c
$ gcc hello.c -o hello -ggdb
$ ls -l
-rwxr-xr-x    1 root     root         4819 Sep 22 00:13 a.out
-rwxr-xr-x    1 root     root        11351 Sep 22 00:14 hello
-rw-r--r--    1 root     root           82 Sep 21 21:53 hello.c

Plik źródłowy został zmieniony. W drugiej linii jest: void main(void).
Kompilator generuje ostrzeżenie (ang. warning).

$ gcc hello.c -o hello
hello.c: In function `main':
hello.c:5: warning: `return' with a value, in function returning void
hello.c:3: warning: return type of `main' is not `int'

Za pomocą przełącznika -w ignorujemy wszystkie ew. ostrzeżenia
kompilatora. Nie jest to jednak dobra praktyka! Ostrzeżenia kompilatora
mogą być preludium do poważnych kłopotów!

$ gcc hello.c -o hello -w
$ ls -l
-rwxr-xr-x    1 root     root         4803 Sep 22 00:26 hello
-rw-r--r--    1 root     root           83 Sep 22 00:25 hello.c

Program uruchomił się, mimo ostrzeżeń kompilatora. W ogólności tak być nie
musi. Program może nie działać lub działać z trudnymi do zdiagnozowania
błędami.

$ ./hello
Hello, world !
$
```

4. Programy z wieloma plikami źródłowymi

Tylko bardzo proste programy składają się wyłącznie z jednego pliku źródłowego. Zwykle tych plików jest wiele, gdyż dzielenie kodów źródłowych na wiele plików jest po prostu wygodne i zwiększa efektywność programowania.

Proces kompilacji programów za pomocą kompilatora `gcc` składa się z czterech etapów: *przetwarzania wstępnego, kompilacji, asemblacji* oraz *łączenia*. Nie wnikając w tej chwili w szczegóły powiedzmy tylko tyle, że efektem pośrednim pomiędzy plikiem źródłowym a plikiem wynikowym jest tzw. kod obiektowy (plik pośredni). Gdy plików źródłowych jest kilka, kilka jest też plików obiektowych. Za pomocą programu scalającego (ang. *linker*) należy połączyć je w jeden plik wykonywalny. Obrazuje to poniższy rysunek:



Każdy z plików źródłowych można oczywiście kompilować oddzielnie, ale chcąc otrzymać jeden wynikowy plik wykonywalny należy skompilować je w taki sposób, aby automatycznie nastąpiło utworzenie plików obiektowych oraz (tu jednego) pliku wynikowego. Wykonanie powyższego zadania z użyciem kompilatora `gcc` jest bardzo proste. Należy po prostu w wierszu poleceń `gcc` podać nazwę wszystkich plików źródłowych, które mają być skompilowane. Reszta zostanie wykonana automatycznie.

Poniżej zamieszczono przykład programu z 3 plikami źródłowymi oraz sposób ich kompilacji. Prosimy o bardzo wnikliwe zapoznanie się z plikami źródłowymi oraz z przykładowym zapisem sesji przy terminalu.

```
// calc.c

#include <stdio.h>
#include <stdlib.h>

// Funkcje nazywam div2, bo istnieje juz w bibliotece standardowe
// funkcja o nazwie div
extern double add (double, double);
extern double sub (double, double);
extern double mul (double, double);
extern double div2(double, double);
// extern gdyż definicje funkcji znajdują się w innym pliku

int main(void)
{
  double a, b;
  char str[80];
  printf("\n\nProsty kalkulator: \n");
  printf("==================\n\n");
  printf ("Podaj pierwsza liczbe (a): ");
  gets(str);
  a = atof(str);
  printf ("Podaj druga liczbe (b): ");
  gets(str);
  b = atof(str);

  printf("\n");
  printf("Liczba a: %8.2f\nLiczba b: %8.2f\n\n", a, b);
  printf("a + b: %8.2f\n", add(a, b));
  printf("a - b: %8.2f\n", sub(a, b));
  printf("a * b: %8.2f\n", mul(a, b));
  printf("a / b: %8.2f\n", div2(a, b));

  printf("\n *** Koniec ***\n\n");
  return 0;
}
```

```
// addsub.c

double add (double a, double b)
{
  return a + b;
}

double sub (double a, double b)
{
  return a - b;
}
```

```
// muldiv.c

double mul (double a, double b)
{
  return a * b;
}

double div2 (double a, double b)
{
  return a / b;
}
```

Na początku mamy tylko 3 pliki źródłowe.

```
$ ls -l
-rw-r--r--    1 root     root          130 Sep 21 23:57 addsub.c
-rw-r--r--    1 root     root          761 Sep 22 00:35 calc.c
-rw-r--r--    1 root     root          130 Sep 21 23:57 muldiv.c
```

Kompilujemy pliki. Kompilator gcc „wie" co z nimi zrobić. Automatycznie stworzy pliki obiektowe oraz połączy je w jeden wynikowy plik wykonywalny. Ostrzeżenia o używaniu funkcji 'gets' wynikają z „podatności" tej funkcji na ataki hacker-ów (przepełnienie bufora).

```
$ gcc calc.c addsub.c muldiv.c -o calc
/tmp/ccnHLJiV.o: In function `main':
/tmp/ccnHLJiV.o(.text+0x3e): the `gets' function is dangerous and should
not be used.
```

W wyniku kompilacji powstaje plik wykonywalny.

```
$ ls -l
-rw-r--r--    1 root     root          130 Sep 21 23:57 addsub.c
-rwxr-xr-x    1 root     root         6049 Sep 22 00:39 calc
-rw-r--r--    1 root     root          761 Sep 22 00:35 calc.c
-rw-r--r--    1 root     root          130 Sep 21 23:57 muldiv.c
```

Tym razem kompilujemy pliki źródłowe tylko do postaci obiektowej. Na dysku powstają pliki z rozszerzeniem .o

```
$ gcc -c calc.c addsub.c muldiv.c
$ ls -l
-rw-r--r--    1 root     root          130 Sep 21 23:57 addsub.c
-rw-r--r--    1 root     root          793 Sep 22 00:39 addsub.o
-rwxr-xr-x    1 root     root         6049 Sep 22 00:39 calc
-rw-r--r--    1 root     root          761 Sep 22 00:35 calc.c
-rw-r--r--    1 root     root         1904 Sep 22 00:39 calc.o
-rw-r--r--    1 root     root          130 Sep 21 23:57 muldiv.c
-rw-r--r--    1 root     root          794 Sep 22 00:39 muldiv.o
```

Utworzone pliki obiektowe łączymy w jeden wynikowy plik wykonywalny. Pliki calc oraz calc2 mają identyczne rozmiary i de facto identyczną strukturę.

```
$ gcc calc.o addsub.o muldiv.o -o calc2
calc.o: In function `main':
calc.o(.text+0x3e): the `gets' function is dangerous and should not be
used.
$ ls -l
-rw-r--r--    1 root     root          130 Sep 21 23:57 addsub.c
-rw-r--r--    1 root     root          793 Sep 22 00:39 addsub.o
```

```
-rwxr-xr-x    1 root     root         6049 Sep 22 00:39 calc
-rw-r--r--    1 root     root          761 Sep 22 00:35 calc.c
-rw-r--r--    1 root     root         1904 Sep 22 00:39 calc.o
-rwxr-xr-x    1 root     root         6049 Sep 22 00:40 calc2
-rw-r--r--    1 root     root          130 Sep 21 23:57 muldiv.c
-rw-r--r--    1 root     root          794 Sep 22 00:39 muldiv.o
```

Oczywiście gdy zmodyfikujemy tylko jeden plik, to nie ma potrzeby kompilowania wszystkich plików źródłowych. Przeanalizuj takie podejście:

```
$ gcc -c calc.c
$ gcc calc.o addsub.o muldiv.o -o calc2
```

Załóżmy, że w katalogu bieżącym znajdują się pliki źródłowe, a w podkatalogu moje-pliki-n *nasze własne pliki nagłówkowe. Niech początek pliku* calc.c *wygląda tak:*

```
// calc.c

#include <stdio.h>
#include <stdlib.h>
#include "moj.h"   // tak zwykło się pisać dla plików „niestandardowych”
...
```

Przykładowa zawartość pliku moj.h*:*

```
// moj.h
#define AUTOR Artur_Gramacki
```

Aby teraz poprawnie skompilować program, należy użyć przełącznika -I

```
$ gcc -c -Imoje-pliki-n calc.c
```

Użycie przełącznika -pedantic bardzo rygorystycznie sprawdza poprawność kodu. W tym przypadku pokazuje, że w pliku moj.k *zapomnieliśmy nacisnąć Enter na końcu wiersza*

```
$ gcc -c -Imoje-pliki-n -pedantic calc.c
In file included from calc.c:5:
moje-pliki-n/moj.h:1: warning: file does not end in newline
```

Uruchamiamy nasz program.

```
$ ./calc


Prosty kalkulator:
==================

Podaj pierwsza liczbe (a): 10
Podaj druga liczbe (b): 20

Liczba a:    10.00
Liczba b:    20.00

a + b:    30.00
a - b:   -10.00
a * b:   200.00
a / b:     0.50

 *** Koniec ***

I to by było na tyle !
```