Wstęp do Informatyki i Programowania
Laboratorium: Lista 3

Przemysław Kobylański

Wprowadzenie

Pliki nagłówkowe

Załóżmy, że program składa się z następujących czterech funkcji:

```
#include <stdio.h>

void g(int n)
{
  printf("g: dostałam liczbę %d\n", n);
}

void h(char s[])
{
  printf("h:  dostałam napis %s\n", s);
}

void f(void)
{
  g(13);
  h(""to jest tekst");
}

int main(void)
{
  f();
  return 0;
}
```

Podzielimy kod źródłowy tak by każda funkcja powyższego programu znalazła się w osobnym pliku.

Aby każda z funkcji mogła być kompilowana osobno (w osobnym pliku), umieścimy w pliku nagłówkowym `funs.h` deklaracje (ale nie definicje) wszystkich funkcji jakie są wywoływane w powyższym programie.

Dzięki temu każdy plik źródłowy zawierający definicję funkcji będzie mógł dołączyć sobie plik `funs.h` i pozna deklaracje innych funkcji zdefiniowanych w innych plikach źródłowych.

W naszym przykładzie mamy plik nagłówkowy:

```
// funs.h

```

```c
void f(void);
void g(int n);
void h(char s[]);
```

oraz następujące pliki źródłowe:

```c
// main.c

#include "funs.h"

int main(void)
{
  f();
  return 0;
}
```

```c
// f.c

#include "funs.h"

void f(void)
{
  g(13);
  h(""to jest tekst");
}
```

```c
// g.c

#include <stdio.h>
#include "funs.h"

void g(int n)
{
  printf("g: dostałam liczbę %d\n", n);
}
```

```c
// h.c

#include <stdio.h>
#include "funs.h"

void h(char s[])
{
  printf("h:  dostałam napis %s\n", s);
}
```

W przypadku gdy pliki nagłówkowe dołączają do siebie dyrektywą `#include` inne pliki nagłówkowe, istnieje niebezpieczeństwo, że pewne pliki nagłówkowe zostaną dołączone więcej niż jeden raz.

Jeśli jest to niepożądane, to na początku pliku nagłówkowego, który nie powinien być dołączony więcej niż raz, powinna zostać wpisana dyrektywa `#pragma once` (to nie jest część standardu języka C ale wiele kompilatorów rozumie tę dyrektywę).

Makefile

W poprzednim punkcie podzieliliśmy program na kilka plików. Przy jego kompilacji skorzystamy z następującego pliku `Makefile`:

```
# Makefile

all: main

main: main.o f.o g.o h.o
    clang -o main main.o f.o g.o h.o

main.o: main.c
    clang -c main.c

f.o: f.c
    clang -c f.c

g.o: g.c
    clang -c g.c

h.o: h.c
    clang -c h.c

clean:
    rm -f main *.o *~
```

Pamiętaj o znakach tabulacji wcinających polecenia `clang` i `rm`.

Jak widać każda z funkcji jest kompilowana do osobnego pliku wynikowego (z rozszerzeniem `.o`) a następnie przekłady funkcji są konsolidowane do pliku wykonywalnego `main`.

Czytając dokumentację polecenia make można dowiedzieć się jak istotnie skrócić powyższy plik `Makefile`.

Oto przebieg kompilacji naszego przykładu:

```
$ make
clang -c main.c
clang -c f.c
clang -c g.c
clang -c h.c
clang -o main main.o f.o g.o h.o
```

I przykładowe uruchomienie:

```
$ ./main
g: dostałam liczbę 13
h:  dostałam napis to jest tekst
```

Poprawmy funkcję `h()` aby drukowała cudzysłowy przed i po drukowanym łańcuchu znaków (zwróć uwagę na dodane znaki `\"`):

```c
// h.c

#include <stdio.h>
#include "funs.h"

void h(char s[])
{
  printf("h:  dostałam napis \"%s\"\n", s);
}
```

Ponownie skompilujmy nasz program:

```
$ make
clang -c h.c
clang -o main main.o f.o g.o h.o
```

Jak widać tym razem została skompilowana tylko funkcja `h()` i nastąpiło ponowne skonsolidowanie programu wykonywalnego `main`.

Wynik uruchomienia zmodyfikowanego programu:

```
$ ./main
g: dostałam liczbę 13
h:  dostałam napis "to jest tekst"
```

Zadanie 1 (4 pkt)

Napisz funkcję `bool palindrom(char napis[])`, której argumentem jest łańcuch znaków ASCII a wartością `true`, gdy argument jest palindromem[1] i `false` w przeciwnym przypadku.

Napisz funkcję `int main(void)` do przetestowania poprawności działania funkcji `palindrom()`.

Każda z funkcji powinna być w osobnym pliku źródłowym. Przygotuj plik `Makefile` do kompilacji i konsolidacji.

Zadanie 2 (8 pkt)

Załóżmy, że funkcja `double f(double x)` oblicza wartość ciągłej funkcji $f(x)$.

Napisz funkcję `double rozwiazanie(double a, double b, double eps)`, która dla zadanych końców przedziału $[a, b]$, gdzie $a < b$, o tej własności, że funkcja $f(x)$ ma na końcach tego przedziału wartości przeciwnego znaku (tzn. $f(a) \cdot f(b) < 0$) znajduje miejsce zerowe funkcji będące rozwiązaniem równania $f(x) = 0$, dla $a < x < b$, z zadaną dokładnością $\varepsilon > 0$ (tzn. znaleziona wartość x nie różni się od faktycznego miejsca zerowego x^* o więcej niż ε, czyli $|x - x^*| \leq \varepsilon$).

[1] Palindrom to takie słowo, które czytane od lewej do prawej i od prawej do lewej jest identyczne. Przykład palindromów: **kajak**, **kobyłamamałybok**.

Przetestuj swój program dla $f(x) = \cos(x/2)$, $a = 2$, $b = 4$ i $\varepsilon \in \{10^{-k} : k = 1, 2, \ldots, 8\}$.

Każda z funkcji programu powinna być w osobnym pliku źródłowym. Przygotuj plik `Makefile` do kompilacji i konsolidacji.

Nie zapomnij podczas konsolidacji o dołączeniu biblioteki funkcji matematycznych.

Uwaga

Przedział $[a, b]$ można podzielić na $\frac{b-a}{\varepsilon}$ kawałków szerokości ε. Zadaniem naszym jest znalezienie tego kawałka spośród nich, w którym znajduje się miejsce zerowe. Nie należy przeglądać kawałków np. od lewej do prawej, gdyż w najgorszym przypadku trzeba sprawdzić wszystkie kawałki.

Wskazówka

Jeśli na końcach przedziału $[a, b]$ funkcja ciągła przyjmuje wartości przeciwnego znaku i wyliczymy punkt $c \in (a, b)$, to miejsce zerowe leży w przedziale $[a, c]$ lub w przedziale $[c, b]$. Zastanów się jak ten fakt wykorzystać do istotnego przyspieszenia poszukiwania miejsca zerowego (zamiast sprawdzać $\frac{b-a}{\varepsilon}$ kawałków można badać $\log_2 \frac{b-a}{\varepsilon}$ przypadków).

Zadanie 3 (8 pkt)

Napisz funkcję `int phi(long int n)` obliczającą funkcję Eulera $\varphi(n)$ równą liczbie liczb z zakresu od 1 do n, które są względnie pierwsze z n (jedynym wspólnym dzielnikiem każdej z nich z n jest 1).

Dopisz funkcję `int main()` umożliwiającą testowanie funkcji `phi()`.

Każda z funkcji powinna być w osobnym pliku źródłowym. Przygotuj plik `Makefile` do kompilacji i konsolidacji.

Zadanie 4 (5 pkt)

Agent porusza się na płaszczyźnie po kracie punktów o współrzędnych całkowitych.

Agent znajdujący się w miejscu (x, y) może wykonać krok na północ i przejdzie na miejsce $(x, y + 1)$.

Agent znajdujący się w miejscu (x, y) może wykonać krok na południe i przejdzie na miejsce $(x, y - 1)$.

Agent znajdujący się w miejscu (x, y) może wykonać krok na wschód i przejdzie na miejsce $(x + 1, y)$.

Agent znajdujący się w miejscu (x, y) może wykonać krok na zachód i przejdzie na miejsce $(x - 1, y)$.

Polecenia:

1. Zdefiniuj w pliku `agents.h` typ `struct agent` do reprezentowania agenta oraz wpisz deklaracje poniższych funkcji.
2. Zdefiniuj funkcję `struct agent newagent(int x, int y)`, która tworzy nowego agenta znajdującego się w zadanym miejscu.
3. Zdefiniuj funkcję `void north(struct agent *a)`, która wykonuje danym agentem krok na północ.
4. Zdefiniuj funkcję `void south(struct agent *a)`, która wykonuje danym agentem krok na południe.
5. Zdefiniuj funkcję `void east(struct agent *a)`, która wykonuje danym agentem krok na wschód.
6. Zdefiniuj funkcję `void west(struct agent *a)`, która wykonuje danym agentem krok na zachód.
7. Zdefiniuj funkcję `double distance(struct agent a1, struct agent a2)`, która oblicza odległość euklidesową między dwoma agentami `a1` i `a2`.

Poniżej pokazano przykładową funkcję `main()`:

```c
#include <stdio.h>
#include "agents.h"

int main(void)
{
  struct agent Bob   = newagent(0, 0);
  struct agent Alice = newagent(3, 3);
  north(&Bob);
  south(&Alice);
  west(&Alice);
  north(&Bob);
  east(&Bob);
  south(&Alice);
  printf("odległość = %f\n", distance(Bob, Alice));
  return 0;
}
```

i wynik jej działania:

```
$ ./main
odległość = 1.414214
```

Każda z funkcji powinna być w osobnym pliku źródłowym. Przygotuj plik `Makefile` do kompilacji i konsolidacji.