Politechnika Białostocka
Wydział Elektryczny
Katedra Elektrotechniki, Energoelektroniki i Elektroenergetyki

Instrukcja
do pracowni specjalistycznej z przedmiotu

Informatyka 2

Kod przedmiotu: **EZ1E3012**

(studia niestacjonarne)

JĘZYK C - PROGRAMY WIELOMODUŁOWE

Numer ćwiczenia

INF26Z

Autor:
dr inż. Jarosław Forenc

Białystok 2021

Spis treści

1. Opis stanowiska ... **3**
1.1. Stosowana aparatura ... 3
1.2. Oprogramowanie ... 3
2. Wiadomości teoretyczne ... **3**
2.1. Programy wielomodułowe w języku C ... 3
2.2. Pliki nagłówkowe ... 8
3. Przebieg ćwiczenia ... **10**
4. Literatura ... **12**
5. Pytania kontrolne ... **13**
6. Wymagania BHP ... **13**

Materiały dydaktyczne przeznaczone dla studentów Wydziału Elektrycznego PB.

© Wydział Elektryczny, Politechnika Białostocka, 2021 (wersja 5.0)

Wszelkie prawa zastrzeżone. Żadna część tej publikacji nie może być kopiowana i odtwarzana w jakiejkolwiek formie i przy użyciu jakichkolwiek środków bez zgody posiadacza praw autorskich.

1. Opis stanowiska

1.1. Stosowana aparatura

Podczas zajęć wykorzystywany jest komputer klasy PC z systemem operacyjnym Microsoft Windows (XP/ 7/10).

1.2. Oprogramowanie

Na komputerach zainstalowane jest środowisko programistyczne Code::Blocks.

2. Wiadomości teoretyczne

2.1. Programy wielomodułowe w języku C

Dotychczas pisane programy w języku C składały się zazwyczaj z jednego pliku (**.c** lub **.cpp**). W rzeczywistości program może składać się z wielu plików z kodem źródłowym (modułów). Podejście takie ma wiele zalet:

- program dzielony jest na niezależne moduły, które mogą być oddzielnie testowane i wykorzystywane w innych programach (biblioteki funkcji),
- poprawia się czytelność kodu programu,
- można wykorzystać w programie funkcje napisane w innych językach programowania.

Sposób podziału programu na moduły zostanie przedstawiony na przykładzie programu zawierającego dwie funkcje:

- **main()** - główna funkcja programu;
- **suma()** - funkcja obliczająca sumę dwóch liczb rzeczywistych.

Program zawierający funkcję obliczającą sumę dwóch liczb rzeczywistych.

```
#include <stdio.h>
float suma(float a, float b)
{
    float y;
    y = a + b;
    return y;
}

int main(void)
{
    float x1 = 10.0, x2 = 20.0, wynik;

    wynik = suma(x1,x2);

    printf("Wynik = %f\n", wynik);

    return 0;
}
```

Powyższy program zostanie podzielony na dwa pliki z kodem źródłowym:

- **funkcje.c** - plik zawierający definicję funkcji **suma()**;
- **myapp.c** - plik zawierający definicję funkcji **main()**.

funkcje.c

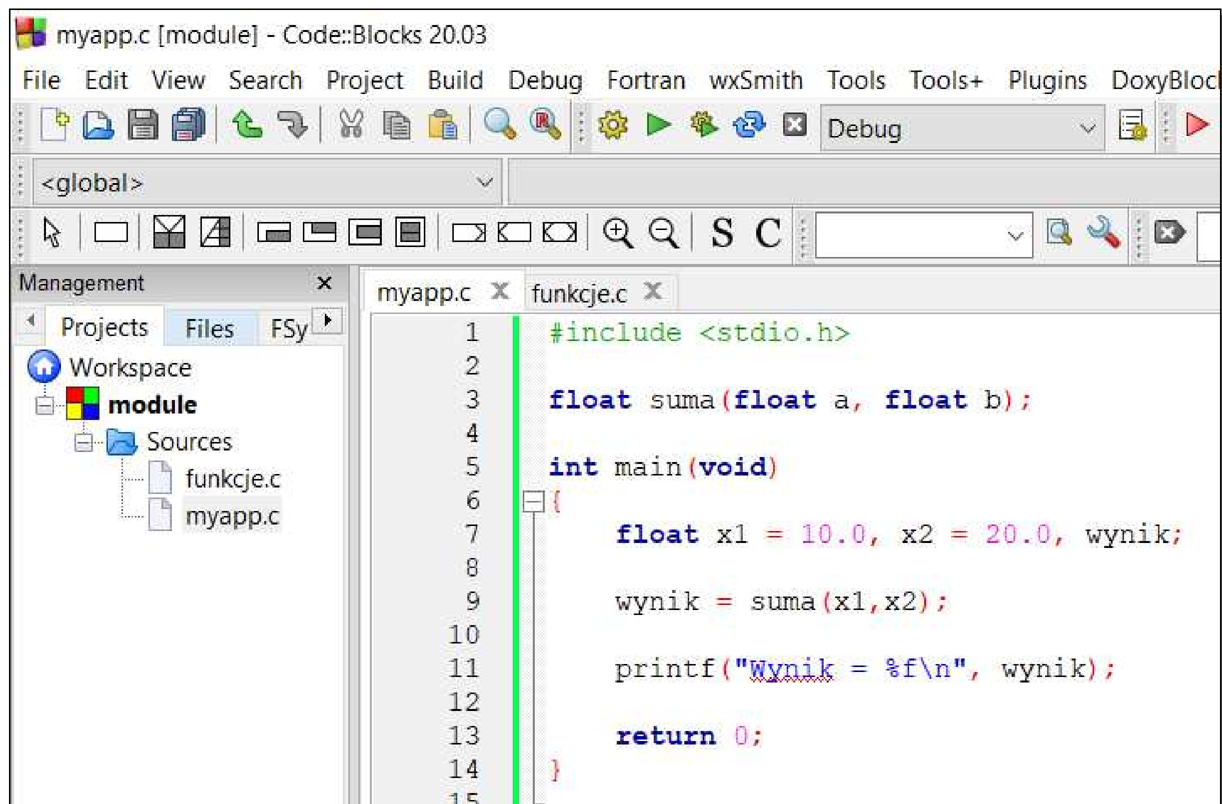
```
float suma(float a, float b)
{
    float y;
    y = a + b;
    return y;
}
```

myapp.c

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

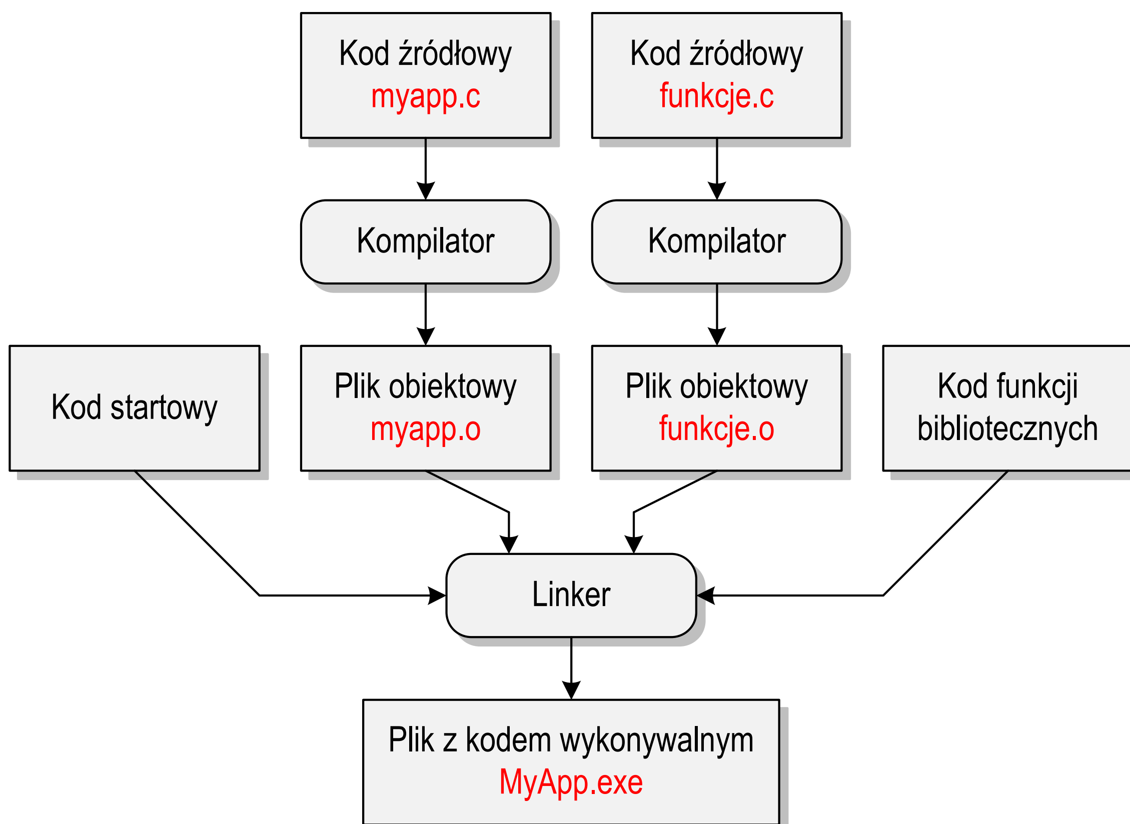
```c
float suma(float a, float b);

int main(void)
{
    float x1 = 10.0, x2 = 20.0, wynik;

    wynik = suma(x1,x2);

    printf("Wynik = %f\n", wynik);

    return 0;
}
```

W przypadku środowiska Code::Blocks stworzenie programu wielomodułowego sprowadza się do dołączenia do projektu kilku plików z kodem źródłowym (Rys. 1). Nowy plik dodajemy wybierając opcję **File → New → Empty file** (**Ctrl-Shift-N**).



Rys. 1. Projekt z programem wielomodułowym

Utworzenie pliku wykonywalnego **exe** odbywa się w standardowy sposób - wybieramy jedną z pozycji znajdujących się w menu głównym **Build**. Operacja ta składa się z dwóch kroków: kompilacji i łączenia (linkowania) (Rys. 2). W trakcie kompilacji kod źródłowy zawarty w plikach **myapp.c** i **funkcje.c** przetwarzany jest na kod maszynowy i zapisywany w plikach obiektowych **myapp.o** i **funkcje.o**. W drugim etapie do plików **.o** dodawany jest kod startowy oraz kod funkcji bibliotecznych, a następnie powstaje plik wykonywalny **exe**.



Rys. 2. Tworzenie pliku wykonywalnego w programie wielomodułowym

Przebieg kompilacji powyższego programu w środowisku Code::Blocks pokazany jest poniżej:

```
-------------- Build: Debug in module (compiler: GNU GCC Compiler)---------------

gcc.exe -Wall -g  -c D:\test\module\funkcje.c -o obj\Debug\funkcje.o
gcc.exe -Wall -g  -c D:\test\module\myapp.c -o obj\Debug\myapp.o
gcc.exe  -o bin\Debug\module.exe obj\Debug\funkcje.o obj\Debug\myapp.o
Output file is bin\Debug\module.exe with size 53.89 KB
```

W pliku **myapp.c** konieczne jest umieszczenie prototypu funkcji **suma()**:

```c
float suma(float a, float b);
```

Brak prototypu funkcji spowoduje wyświetlenia ostrzeżenia podczas kompilacji:

```
||=== Build: Debug in module (compiler: GNU GCC Compiler) ===|
myapp.c||In function 'main':|
myapp.c|9|warning: implicit declaration of function 'suma' [-Wimplicit-function-declaration]|
||=== Build finished: 0 error(s), 1 warning(s) (0 minute(s), 0 second(s)) ===|
```

oraz niepoprawną pracę skompilowanego programu.

Jeśli w pliku **myapp.c** będzie znajdowała się deklaracja funkcji **suma()**, ale w pliku **funkcje.c** nie będzie jej definicji, to błąd wystąpi nie na etapie kompilacji, ale na etapie łączenia.

Czasami w pisanym programie występuje konieczność odwołania się do zmiennej, która została zadeklarowana w innym pliku z kodem źródłowym. Aby odwołanie takie było możliwe zmienna ta powinna być przede wszystkim zadeklarowana jako **globalna** (tzn. jej deklaracja powinna znajdować się poza ciałami funkcji). W pliku, w którym chcemy z niej skorzystać, należy umieścić ponownie jej deklarację, ale poprzedzoną słowem **extern**. Sytuację taką pokazuje poniższy przykład programu składającego się z plików **prog1.c** i **prog2.c**.

prog1.c

```c
int x = 10;     // definicja zmiennej globalnej
```

prog2.c

```c
#include <stdio.h>

extern int x;     // deklaracja zmiennej globalnej
```

```c
int main(void)
{
    printf("x = %d\n", x);

    return 0;
}
```

Wynik działania programu:

```
x = 10
```

2.2. Pliki nagłówkowe

W przypadku dużej liczby zmiennych oraz funkcji pochodzących z innych modułów „ręczne” umieszczanie deklaracji zmiennych ze słowem **extern** i deklaracji funkcji (prototypów) może być uciążliwe. W praktyce deklaracje takie umieszcza się w plikach nagłówkowych (z rozszerzeniem **.h**). Następnie plik taki dołączany jest dyrektywą preprocesora **#include** do wszystkich modułów, w których występują odwołania do tych zmiennych lub funkcji. Sytuację taką pokazuje poniższy program składający się z trzech plików z kodem źródłowym:

- **funkcje.h** - plik zawierający deklarację funkcji **suma()** i zmiennej **x**;
- **funkcje.c** - plik zawierający definicję funkcji **suma()** i zmiennej **x**;
- **myapp.c** - plik zawierający definicję funkcji **main()**,wywołanie funkcji **suma()** i odwołanie do zmiennej **x**.

funkcje.h

```c
float suma(float a, float b);
extern int x;
```

funkcje.c

```c
int x = 10;
```

```c
float suma(float a, float b)
{
    float y;
    y = a + b;
    return y;
}
```

myapp.c

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

int main(void)
{
    float x1 = 10.0, x2 = 20.0, wynik;

    wynik = suma(x1,x2);

    printf("Wynik = %f\n", wynik);
    printf("x = %d\n", x);

    return 0;
}
```

W rozbudowanych programach może zdarzyć się tak, że ten sam plik nagłówkowy zostanie dołączony do modułu kilka razy. Może spowodować to błąd kompilacji. Rozwiązaniem tego problemu jest zastosowanie dyrektyw kompilacji warunkowej: **#ifndef** i **#endif**. Dyrektywa **#ifndef** sprawdza czy występujący po niej identyfikator został zdefiniowany. Jeśli tak, to kod znajdujący się do dyrektywy **#endif** jest pomijany podczas kompilacji. Nowa wersja pliku nagłówkowego **funkcje.h** jest pokazana poniżej.

funkcje.h

```c
#ifndef _FUNKCJE_H_
#define _FUNKCJE_H_
```

```
float suma(float a, float b);
extern int x;

#endif
```

Przy pierwszym dołączaniu pliku nagłówkowego **funkcje.h** jeszcze nie istnieje identyfikator **_FUNKCJE_H_** więc jest on tworzony (dyrektywa **#define**), a następnie deklarowana jest funkcja **suma()** i zmienna **x**. Podczas kolejnego dołączania tego samego pliku nagłówkowego identyfikator **_FUNKCJE_H_** będzie już istniał więc cały kod występujący po dyrektywie **#ifndef** aż do dyrektywy **#endif** zostanie pominięty.

3. Przebieg ćwiczenia

Na pracowni specjalistycznej należy wykonać wybrane zadania wskazane przez prowadzącego zajęcia. W różnych grupach mogą być wykonywane różne zadania.

1. Napisz program zawierający funkcje wykonujące operacje na wektorze liczb całkowitych:
   a) **wektor_generuj()** - funkcja zapisująca do wektora wygenerowane pseudolosowo liczby z zakresu ⟨**a, b**⟩; argumenty funkcji: wektor, rozmiar wektora, zakres generowanych liczb;
   b) **wektor_drukuj()** - funkcja wyświetlająca elementy wektora w jednym wierszu; argumenty funkcji: wektor, rozmiar wektora;
   c) **wektor_suma()** - funkcja zwracająca sumę elementów wektora; argumenty funkcji: wektor, rozmiar wektora;
   d) **wektor_max()** - funkcja zwracająca wartość największego elementu wektora; argumenty funkcji: wektor, rozmiar wektora;
   e) **wektor_ile_x()** - funkcja zwracająca liczbę elementów wektora równych wartości **x**; argumenty funkcji: wektor, rozmiar wektora, liczba **x**;

f) **wektor_sortuj() -** funkcja sortująca elementy wektora dowolną metodą w kolejności rosnącej; argumenty funkcji: wektor, rozmiar wektora.

Rozmiar wektora wczytaj z klawiatury. Pamięć na wektor przydziel dynamicznie. Wywołaj wszystkie funkcje zdefiniowane w programie. Kod źródłowy programu zapisz w trzech plikach: **wektor.c** (definicje funkcji), **wektor.h** (deklaracje funkcji), **main.c** (definicja funkcji **main**). Zabezpiecz plik nagłówkowy przed wielokrotnym dołączaniem kodu.

2. Napisz program zawierający funkcje wykonujące operacje na macierzy liczb całkowitych:

   a) **macierz_generuj()** - funkcja zapisująca do macierzy wygenerowane pseudolosowo liczby z zakresu ⟨**a, b**⟩; argumenty funkcji: macierz, liczba wierszy i liczba kolumn macierzy, zakres generowanych liczb;

   b) **macierz_drukuj() -** funkcja wyświetlająca elementy macierzy z podziałem na wiersze i kolumny; argumenty funkcji: macierz, liczba wierszy i kolumn macierzy;

   c) **macierz_suma() -** funkcja zwracająca sumę elementów macierzy; argumenty funkcji: macierz, liczba wierszy i kolumn macierzy;

   d) **macierz_max() -** funkcja zwracająca wartość największego elementu macierzy; argumenty funkcji: macierz, liczba wierszy i kolumn macierzy;

   e) **macierz_transpozycja() -** funkcja dokonująca transpozycji macierzy; argumenty funkcji: macierz, liczba wierszy i kolumn macierzy.

   Liczbę wierszy i kolumn macierzy wczytaj z klawiatury. Pamięć na macierz przydziel dynamicznie. Wywołaj wszystkie funkcje zdefiniowane w programie. Kod źródłowy programu zapisz w trzech plikach: **macierz.c** (definicje funkcji), **macierz.h** (deklaracje funkcji), **main.c** (definicja funkcji **main**). Zabezpiecz plik nagłówkowy przed wielokrotnym dołączaniem kodu.

3. Napisz program zawierający funkcje wykonujące operacje na liczbach zespolonych:

a) **complex_drukuj()** - funkcja wyświetlająca liczbę zespoloną; argumenty funkcji: liczba zespolona;
b) **complex_suma()** - funkcja obliczająca sumę dwóch liczb zespolonych: argumenty funkcji: dwie liczby zespolone;
c) **complex_roznica()** - funkcja obliczająca różnicę dwóch liczb zespolonych: argumenty funkcji: dwie liczby zespolone;
d) **complex_iloraz()** - funkcja obliczająca iloraz dwóch liczb zespolonych: argumenty funkcji: dwie liczby zespolone;
e) **complex_iloczyn()** - funkcja obliczająca iloczyn dwóch liczb zespolonych: argumenty funkcji: dwie liczby zespolone;
f) **complex_modul()** - funkcja obliczająca moduł liczby zespolonej: argumenty funkcji: liczba zespolona;
g) **complex_argument_st()** - funkcja obliczająca argument (kąt) liczby zespolonej w stopniach: argumenty funkcji: liczba zespolona;
h) **complex_argument_rad()** - funkcja obliczająca argument (kąt) liczby zespolonej w radianach: argumenty funkcji: liczba zespolona.

Do przechowywania liczby zespolonej zastosuj strukturę. Wywołaj wszystkie funkcje zdefiniowane w programie. Kod źródłowy programu zapisz w trzech plikach: **zespolone.c** (definicje funkcji), **zespolone.h** (deklaracje funkcji), **main.c** (definicja funkcji **main**). Zabezpiecz plik nagłówkowy przed wielokrotnym dołączaniem kodu.

4. Literatura

[1] Prata S.: Język C. Szkoła programowania. Wydanie VI. Helion, Gliwice, 2016.

[2] Kernighan B.W., Ritchie D.M.: Język ANSI C. Programowanie. Wydanie II. Helion, Gliwice, 2010.

[3] Deitel P.J., Deitel H.: Język C. Solidna wiedza w praktyce. Wydanie VIII. Helion, Gliwice, 2020.

[4] Kochan S.G.: Język C. Kompendium wiedzy. Wydanie IV. Helion, Gliwice, 2015.

[5] King K.N.: Język C. Nowoczesne programowanie. Wydanie II. Helion, Gliwice, 2011.

[6] http://www.cplusplus.com/reference/clibrary - C library - C++ Reference

[7] https://cpp0x.pl/dokumentacja/standard-C/1 - Standard C

[8] https://www.codeblocks.org/ - Code::Blocks

5. Pytania kontrolne

1. Wyjaśnij, po co stosuje się programy wielomodułowe?
2. W jaki sposób w programach wielomodułowych można odwoływać się do funkcji i zmiennych zdefiniowanych w innych modułach?
3. W jaki sposób można zabezpieczyć plik nagłówkowy przed wielokrotnym dołączaniem jego kodu?

6. Wymagania BHP

Warunkiem przystąpienia do praktycznej realizacji ćwiczenia jest zapoznanie się z instrukcją BHP i instrukcją przeciw pożarową oraz przestrzeganie zasad w nich zawartych.

W trakcie zajęć laboratoryjnych należy przestrzegać następujących zasad.

- Sprawdzić, czy urządzenia dostępne na stanowisku laboratoryjnym są w stanie kompletnym, nie wskazującym na fizyczne uszkodzenie.
- Jeżeli istnieje taka możliwość, należy dostosować warunki stanowiska do własnych potrzeb, ze względu na ergonomię. Monitor komputera ustawić w sposób zapewniający stałą i wygodną obserwację dla wszystkich członków zespołu.
- Sprawdzić prawidłowość połączeń urządzeń.

- Załączenie komputera może nastąpić po wyrażeniu zgody przez prowadzącego.
- W trakcie pracy z komputerem zabronione jest spożywanie posiłków i picie napojów.
- W przypadku zakończenia pracy należy zakończyć sesję przez wydanie polecenia wylogowania. Zamknięcie systemu operacyjnego może się odbywać tylko na wyraźne polecenie prowadzącego.
- Zabronione jest dokonywanie jakichkolwiek przełączeń oraz wymiana elementów składowych stanowiska.
- Zabroniona jest zmiana konfiguracji komputera, w tym systemu operacyjnego i programów użytkowych, która nie wynika z programu zajęć i nie jest wykonywana w porozumieniu z prowadzącym zajęcia.
- W przypadku zaniku napięcia zasilającego należy niezwłocznie wyłączyć wszystkie urządzenia.
- Stwierdzone wszelkie braki w wyposażeniu stanowiska oraz nieprawidłowości w funkcjonowaniu sprzętu należy przekazywać prowadzącemu zajęcia.
- Zabrania się samodzielnego włączania, manipulowania i korzystania z urządzeń nie należących do danego ćwiczenia.
- W przypadku wystąpienia porażenia prądem elektrycznym należy niezwłocznie wyłączyć zasilanie stanowiska. Przed odłączeniem napięcia nie dotykać porażonego.