Fundusze Europejskie
Rzeczpospolita Polska
Dofinansowane przez Unię Europejską

Inżynier 5.0 - kształcenie na potrzeby gospodarki
Projekt współfinansowany przez Unię Europejską w ramach programu Fundusze Europejskie dla Rozwoju Społecznego 2021-2027.
Nr umowy o dofinansowanie: FERS.01.05-IP.08-0285/23-00.

Wprowadzenie do programowania

Instrukcja do ćwiczenia laboratoryjnego nr 2

Temat: Instrukcje sterujące i funkcje

Politechnika Gdańska
Wydział Elektroniki, Telekomunikacji i Informatyki
Katedra Inżynierii Biomedycznej

Opracował: mgr inż. Dmytro Tkachenko

Spis treści

1 Wprowadzenie

Celem tego laboratorium jest utrwalenie wiedzy z wykładów oraz praktyczne zapoznanie się z instrukcjami sterującymi i funkcjami.

1.1 Instrukcje sterujące

Kiedy program jest uruchamiany, procesor (CPU) rozpoczyna wykonanie na początku funkcji `main()`, wykonuje pewną liczbę instrukcji (domyślnie w kolejności sekwencyjnej), a program kończy działanie na końcu funkcji `main()`.

Rozważ następujący program:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("Enter an integer: ");
6
7      int x;
8      scanf("%d", &x);
9
10     printf("Your number %d\n", x);
11
12     return 0;
13 }
```

Ścieżka wykonania tego programu obejmuje linie 5, 7, 8, 10 i 12, w dokładnie tej kolejności. Jest to przykład programu liniowego. Programy liniowe za każdym razem, gdy są uruchamiane, przechodzą tę samą ścieżkę (wykonują te same instrukcje w tej samej kolejności).

Jednak często nie jest to to, czego chcemy. Na przykład, jeśli poprosimy użytkownika o wprowadzenie danych, a użytkownik wprowadzi coś nieprawidłowego, idealnie chcielibyśmy poprosić go o dokonanie kolejnego wyboru. Nie jest to możliwe w programie liniowym. W rzeczywistości użytkownik może wielokrotnie wprowadzać nieprawidłowe dane, więc liczba prób, jaką będziemy potrzebować, aby poprosić go o ponowny wybór, nie jest znana aż do momentu uruchomienia programu. Na szczęście C zapewnia szereg różnych instrukcji sterujących, które pozwalają zmienić normalną ścieżkę wykonania programu (spójrz Tabela 1).

1.1.1 Instrukcja `if`

Najbardziej podstawowym rodzajem instrukcji warunkowej w C jest instrukcja `if`. Instrukcja `if` przyjmuje postać:

```
if (condition)
    true_statement;
```

lub z opcjonalną instrukcją `else`:

Kategoria	Znaczenie	Zaimplementowane w C przez
Instrukcje warunkowe	Powodują wykonanie sekwencji kodu tylko wtedy, gdy spełniony jest pewien warunek.	`if`, `else`, `switch`
Skoki	Nakazują procesorowi rozpoczęcie wykonywania instrukcji w innej lokalizacji.	`goto, break, continue`
Wywołania funkcji	Skok do innej lokalizacji i powrót.	wywołanie funkcji, `return`
Pętle	Wielokrotnie wykonują pewną sekwencję kodu zero lub więcej razy, aż spełniony zostanie pewien warunek.	`while, do-while, for`
Zatrzymania	Kończą działanie programu.	`exit()`, itd.

Tabela 1: Kategorie instrukcji sterujących w C

```
if (condition)
    true_statement;
else
    false_statement;
```

Jeśli warunek (`condition`) jest spełniony, wykonuje się `true_statement`. Jeśli warunek nie jest spełniony i istnieje opcjonalna instrukcja else, wykonuje się `false_statement`.

Poniżej prosty program, który używa instrukcji **`if`** z opcjonalną instrukcją **`else`**:

```c
#include <stdio.h>

int main()
{
    printf("Enter a number: ");
    int x;
    scanf("%d", &x);

    if (x > 10) {
        printf("%d is greater than 10\n", x);
    }
    else {
        printf("%d is not greater than 10\n", x);
    }

    return 0;
}
```

Uruchom program i sprawdź jego działanie.

Instrukcje **`if`** oraz **`else`** mogą mieć wiele instrukcji warunkowych.

Nawiasy klamrowe – kiedy są potrzebne i dlaczego to ważne?

W języku C nawiasy klamrowe {} grupują kilka instrukcji w jeden blok. W przypadku instrukcji `if` (lub `else`) możesz ich nie używać tylko wtedy, gdy po nich występuje jedna pojedyncza instrukcja.

Brak nawiasów klamrowych może prowadzić do błędów logicznych, które są trudne do zauważenia. Zobacz przykład poniżej:

```
if (x > 10)
printf("x is greater than 10\n");
printf("This message is always printed!\n");
```

Na pierwszy rzut oka może się wydawać, że obie linie należą do bloku `if`, ale tak nie jest. W rzeczywistości tylko pierwsze wywołanie `printf` jest zależne od warunku `x` > 10, a druga instrukcja wykona się zawsze, niezależnie od wartości `x`.

Aby uniknąć takich błędów, w praktyce zaleca się zawsze używać nawiasów klamrowych {}, nawet gdy ciało instrukcji składa się tylko z jednej linii. To poprawia czytelność kodu i zapobiega przypadkowemu wprowadzeniu błędów podczas modyfikacji programu.

1.1.2 Instrukcja `switch`

Chociaż możliwe jest łączenie wielu instrukcji `if-else` w łańcuch, jest to zarówno trudne do czytania, jak i nieefektywne. Rozważ następujący program:

```
#include <stdio.h>

int main()
{
    int x = 2;
    if (x == 1)
        printf("One");
    else if (x == 2)
        printf("Two");
    else if (x == 3)
        printf("Three");
    else
        printf("Unknown");
    printf("\n");

    return 0;
}
```

Zmienna `x` będzie sprawdzana do trzech razy w zależności od przekazanej wartości (co jest nieefektywne), a my musimy być pewni, że za każdym razem sprawdzana jest właśnie zmienna `x` (a nie jakaś inna zmienna).

Język C zapewnia alternatywną instrukcję warunkową zwaną instrukcją `switch`, która jest specjalnie do tego przeznaczona. Oto ten sam program, co powyżej, z użyciem instrukcji `switch`:

```c
#include <stdio.h>

int main()
{
    int x = 2;
    switch (x)
    {
    case 1:
        printf("One");
        return;
    case 2:
        printf("Two");
        return;
    case 3:
        printf("Three");
        return;
    default:
        printf("Unknown");
        return;
    }
    printf("\n");

    return 0;
}
```

Ideą stojącą za instrukcją **switch** jest prosta: wyrażenie (czasami nazywane warunkiem) jest ewaluowane, aby wyprodukować wartość.

Następnie występuje jedna z następujących sytuacji:

- Jeśli wartość wyrażenia jest równa wartości po którymkolwiek z etykiet **case**, instrukcje po pasującej etykiecie **case** są wykonywane.
- Jeśli nie zostanie znaleziona pasująca wartość i istnieje etykieta **default**, instrukcje po etykiecie **default** są wykonywane.
- Jeśli nie zostanie znaleziona pasująca wartość i nie ma etykiety **default**, instrukcja **switch** jest pomijana.

1.1.3 Instrukcja `while`

Instrukcja **while** (nazywana również pętlą **while**) jest najprostszym z trzech typów pętli dostępnych w c i ma definicję bardzo podobną do instrukcji **if**:

```c
while (condition)
    statement;
```

Kiedy instrukcja **while** jest wykonywana, wyrażenie `condition` jest ewaluowane. Jeśli warunek jest spełniony (ewaluuje do true), powiązana instrukcja jest wykonywana.

Jednak, w przeciwieństwie do instrukcji **if**, gdy instrukcja zakończy wykonanie, sterowanie wraca na początek instrukcji **while** i proces jest powtarzany. Oznacza to, że instrukcja **while** będzie kontynuować zapętlanie tak długo, jak warunek będzie spełniony.

Poniżej przykład z prostą pętlą **while**, która wypisuje wszystkie liczby od 1 do 10:

```
#include <stdio.h>

int main()
{
    int count = 1; // inicjalizacja zmiennej
    while (count <= 10) // rób dopóki 'count' mniejszy lub równy 10
    {
        printf("%d ", count);
        ++count; // zwiększ o jeden
    }

    printf("done!\n");

    return 0;
}
```

1.1.4 Instrukcja do-while

Instrukcja do-while ma składnię jak poniżej:

```
do
    statement; // może być pojedynczą instrukcją lub instrukcją złożoną
while (condition);
```

Instrukcja **do-while** jest konstrukcją pętli, która działa podobnie jak pętla **while**, z tą różnicą, że instrukcja zawsze wykonuje się przynajmniej raz. Po wykonaniu instrukcji, pętla **do-while** sprawdza warunek. Jeśli warunek jest spełniony, ścieżka wykonania wraca na początek pętli **do-while** i wykonuje ją ponownie.

Przykład z użyciem pętli **do-while**:

```
#include <stdio.h>

int main()
{
    // selection musi być zadeklarowane poza pętlą do-while,
    // abyśmy mogli później użyć tę zmienną
    int selection = 0;
    do
    {
        printf("Please make a selection: \n");
        printf("1) Addition\n");
        printf("2) Subtraction\n");
```

```
13          printf("3) Multiplication\n");
14          printf("4) Division\n");
15          scanf("%d", &selection);
16      }
17      while (selection < 1 || selection > 4);
18
19      // zrób coś z wyborem tutaj
20      // na przykład instrukcja switch
21
22      printf("You selected option #%d\n", selection);
23
24      return 0;
25  }
```

Zwróć uwagę na linię 17 w kodzie powyżej. Tutaj użyty został operator logiczny OR (||). Operator || zwraca wartość prawda (true), jeśli co najmniej jeden z jego operandów jest prawdziwy. W naszym przypadku pętla będzie się powtarzać, dopóki `selection` jest mniejsze niż 1 lub większe niż 4. Oznacza to, że pętla zakończy się tylko wtedy, gdy `selection` przyjmie wartość z przedziału od 1 do 4 włącznie.

1.1.5 Instrukcja `for`

Zdecydowanie najczęściej wykorzystywaną instrukcją pętli w C jest instrukcja **`for`**. Instrukcja **`for`** (zwana również pętlą **`for`**) jest preferowana, gdy mamy oczywistą zmienną pętli, ponieważ pozwala nam w łatwy i zwięzły sposób definiować, inicjalizować, testować i zmieniać wartość zmiennych pętli.

Instrukcja **`for`** wygląda dość prosto w formie abstrakcyjnej:

```
for (init-statement; condition; end-expression)
   statement;
```

Najłatwiejszym sposobem na początku zrozumienia, jak działa instrukcja **`for`**, jest przekształcenie jej w równoważną instrukcję **`while`**:

```
{ // instrukcja złożona (block)
    init-statement; // instrukcja używana do definicji zmiennych używanych w pętli
    while (condition)
    {
        statement;
        end-expression; // używane do modyfikacji zmiennej pętli
    }
} // zmienne zdefiniowane wewnątrz pętli tracą zakres tutaj
```

Kolejność wykonywania różnych części instrukcji for jest następująca:

- `Init-statement` (instrukcja inicjująca)
- `Condition` (warunek) (jeśli jest niespełniony, pętla kończy się tutaj)

- Instrukcje w środku pętli
- `End-expression` (wyrażenie kończące) (następnie skok z powrotem do warunku)

Zauważ, że `end-expression` wykonuje się po instrukcji pętli, a następnie warunek jest ponownie sprawdzany.

Poniżej przykładowa pętla `for` wypisująca liczby od 1 do 10:

```
#include <stdio.h>

int main()
{
    int i;
    for (i = 0; i <= 10; ++i)
        printf("%d ", i);

    printf("\n");

    return 0;
}
```

Najpierw deklarujemy zmienną pętli o nazwie `i` i inicjalizujemy ją wartością `1`.

Potem, `i <= 10` jest ewaluowane, a ponieważ `i` wynosi `1`, to zwraca `true`. W konsekwencji instrukcja jest wykonywana, co wypisuje `1` i spację.

Wreszcie, `++i` jest wykonywane, co inkrementuje `i` do `2`. Następnie pętla wraca do drugiego kroku.

Teraz warunek `i <= 10` jest sprawdzany ponownie. Ponieważ `i` ma wartość `2`, to zwraca `true`, więc pętla iteruje ponownie. Instrukcja wypisuje `2` i spację, a `i` jest inkrementowane do `3`. Pętla kontynuuje iterowanie, aż w końcu `i` będzie równe `11`. W tym momencie `i <= 10` zwraca `false` i pętla kończy działanie.

1.1.6 Instrukcje `break` oraz `continue`

W języku C instrukcje `break` i `continue` są elementami sterującymi przepływem wykonania w pętlach oraz instrukcjach `switch`. Umożliwiają one bardziej elastyczne zarządzanie przebiegiem programu.

Słowo kluczowe `break` służy do natychmiastowego przerwania działania:

- pętli (`for`, `while`, `do-while`) – powoduje wyjście z niej, a dalszy kod po pętli jest wykonywany normalnie
- instrukcji `switch` – zapobiega tzw. przeciekaniu (fall-through) do kolejnych etykiet `case`.

Przykład użycia `break` w pętli:

```
#include <stdio.h>

int main() {
```

```
    for (int i = 1; i <= 10; ++i) {
        if (i == 5)
            break;
        printf("%d ", i);
    }
    printf("\nKoniec pętli.\n");
    return 0;
}
```

Wynik:

```
1 2 3 4
Koniec pętli.
```

Pętla kończy się, gdy `i` osiągnie wartość 5.

Przykład użycia **`break`** w **`switch`**:

```
switch (x) {
    case 1:
        printf("Jeden\n");
        break;
    case 2:
        printf("Dwa\n");
        break;
    default:
        printf("Inna liczba\n");
        break;
}
```

Bez **`break`** program wykonywałby instrukcje we wszystkich kolejnych **`case`**, nawet jeśli warunek nie jest spełniony.

Słowo kluczowe **`continue`** przerywa bieżącą iterację pętli i przechodzi od razu do następnej iteracji. Program wraca do sprawdzenia warunku pętli (lub do `end-expression` w pętli **`for`**).

Przykład użycia **`continue`**:

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; ++i) {
        if (i == 3)
            continue;
        printf("%d ", i);
    }
    printf("\nKoniec.\n");
    return 0;
}
```

Wynik:

```
1 2 4 5
Koniec.
```

1.1.7 Zadania

Zadanie 1.1

Poniżej pętla ma wypisywać wszystkie liczby od 9 do 0. Co jest z nią nie tak?

```
int i;
for (i = 9; i > 0; --i) {
    printf("%d ", i);
}
```

Zadanie 1.2

Napisz program, który wypisze litery od a do z wraz z ich kodami ASCII. Użyj zmiennej sterującej pętlą typu `char`.

Zadanie 1.3

Napisz program z pętlą `for`, który wypisze następujący wynik:

```
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
```

Zadanie 1.4

Zmień program z poprzedniego zadania tak, aby wypisał następujący wynik:

```
        1
      2 1
    3 2 1
  4 3 2 1
5 4 3 2 1
```

1.2 Funkcje

1.2.1 Wstęp do funkcji w C

Funkcja to wielokrotnie używany ciąg instrukcji zaprojektowany do wykonania określonego zadania.

Wiesz już, że każdy wykonywalny program musi mieć funkcję o nazwie `main()` (to w niej program rozpoczyna wykonanie po uruchomieniu). Jednak gdy programy stają się coraz dłuższe, umieszczanie całego kodu wewnątrz funkcji `main()` staje się coraz trudniejsze do zarządzania. Funkcje zapewniają nam sposób na podzielenie naszych programów na małe, modułowe fragmenty, które są łatwiejsze do zorganizowania, przetestowania i użycia.

Program wykonuje instrukcje sekwencyjnie wewnątrz jednej funkcji, gdy napotka wywołanie funkcji. Wywołanie funkcji nakazuje procesorowi przerwanie bieżącej funkcji i wykonanie innej funkcji. Procesor zasadniczo "umieszcza zakładkę"w bieżącym punkcie wykonania, wykonuje funkcję o nazwie podanej w wywołaniu, a następnie wraca do punktu,

który zaznaczył zakładką, i wznawia wykonanie.

Najpierw zacznijmy od najbardziej podstawowej składni definiowania funkcji zdefiniowanej przez użytkownika. Przez kilka kolejnych lekcji wszystkie funkcje zdefiniowane przez użytkownika będą przyjmować następującą formę:

```
// Poniższa linia kodu jest nagłówkiem funkcji
// Mówi kompilatorowi o istnieniu funkcji
returnType functionName()
{
    // To jest ciało funkcji (mówi kompilatorowi, co funkcja robi)
}
```

`returnType` to jest typ który będzie zwracany przez funkcję. Aby wywołać funkcję, używamy nazwy funkcji z nawiasami (np. functionName())

```
#include <stdio.h>

// Definicja funkcji doPrint()
// doPrint() jest wywoływaną funkcją w tym przykładzie
void doPrint()
{
    printf("In doPrint()\n");
}

// Definicja funkcji main()
int main()
{
    printf("Starting main()\n");
    // Przerwanie main() przez wywołanie funkcji doPrint().
    // main() jest funkcją wywołującą.
    doPrint();
    // Ta instrukcja jest wykonywana po zakończeniu doPrint()
    printf("Ending main()\n");

    return 0;
}
```

1.2.2 Wartości zwracane przez funkcje

Kiedy piszesz funkcję, decydujesz, czy twoja funkcja zwróci wartość, czy nie. Aby zwrócić wartość, potrzebne są dwie rzeczy.

Po pierwsze, twoja funkcja musi wskazać, jaki typ wartości będzie zwracany. Robi się to przez ustawienie typu zwracanego funkcji, który jest typem zdefiniowanym przed nazwą funkcji. Gdy nie chcemy, żeby nasza funkcja coś zwracała, ustawiamy typ zwracany jako `void()` (co oznacza, że żadna wartość nie zostanie zwrócona do wywołującej funkcji). Zauważ, że to nie określa, jaka konkretna wartość jest zwracana — określa tylko, jaki typ wartości będzie zwracany.

Po drugie, wewnątrz funkcji, która ma zwrócić wartość, używamy instrukcji **return**, aby wskazać konkretną wartość zwracaną do wywołującej. Instrukcja **return** składa się ze słowa kluczowego **return**, po którym następuje wyrażenie, zakończone średnikiem.

Funkcja zwracająca wartość będzie zwracać wartość za każdym razem, gdy zostanie wywołana.

Spójrzmy na prostą funkcję, która zwraca wartość całkowitą, i przykładowy program, który ją wywołuje:

```
#include <stdio.h>

// int jest typem zwracanym
// Typ zwracany int oznacza, że funkcja zwróci jakąś wartość całkowitą
// (konkretna wartość nie jest tu określona)
int returnFive()
{
    // instrukcja return dostarcza wartość, która zostanie zwrócona
    return 5; // zwróć wartość 5 z powrotem do funkcji wywołującej
}

int main()
{
    printf("%d\n", returnFive()); // wypisuje 5
    printf("%d\n", returnFive() + 2); // wypisuje 7

    // okay: wartość 5 jest zwracana,
    // ale jest ignorowana, ponieważ main() nic z nią nie robi
    returnFive();

    return 0;
}
```

1.2.3 Parametry i argumenty funkcji

W wielu przypadkach przydatne jest przekazywanie informacji do wywoływanej funkcji, aby miała dane do pracy. Parametr funkcji to zmienna używana w nagłówku funkcji. Parametry funkcji działają niemal identycznie jak zmienne zdefiniowane wewnątrz funkcji, ale z jedną różnicą: są inicjalizowane wartością dostarczoną przez wywołującego funkcję.

Parametry funkcji są definiowane w nagłówku funkcji poprzez umieszczenie ich między nawiasami po nazwie funkcji, przy czym wiele parametrów jest oddzielonych przecinkami.

Oto kilka przykładów funkcji z różną liczbą parametrów:

```
// Ta funkcja nie przyjmuje parametrów
// Nie polega na wywołującym w żaden sposób
void doPrint()
{
    printf("In doPrint()\n");
}

```

```c
// Ta funkcja przyjmuje jeden parametr całkowity o nazwie x
// Wywołujący poda wartość x
void printValue(int x)
{
    printf("%d\n", x);
}

// Ta funkcja ma dwa parametry całkowite, jeden o nazwie x i jeden o nazwie y
// Wywołujący poda wartości zarówno x, jak i y
int add(int x, int y)
{
    return x + y;
}
```

Argument to wartość przekazywana od wywołującego do funkcji podczas wywołania funkcji:

```c
doPrint();          // to wywołanie nie ma argumentów
printValue(6);      // 6 jest argumentem przekazanym do funkcji printValue()
add(2, 3);          // 2 i 3 są argumentami przekazanymi do funkcji add()
```

1.2.4 Zadania

Zadanie 2.1

Przeanalizuj (nie kompiluj) każdy z poniższych programów. Określ, co program wypisze, lub czy program wygeneruje błąd kompilacji.

Przykład A

```c
#include <stdio.h>

int return7()
{
    return 7;
}

int return9()
{
    return 9;
}

int main()
{
    printf("%d\n", return7() + return9());
    return 0;
}
```

Przykład B

```c
#include <stdio.h>

int return7()
{
    return 7;

    int return9()
    {
        return 9;
    }
}

int main()
{
    printf("%d\n", return7() + return9());
    return 0;
}
```

Przykład C

```c
#include <stdio.h>

int add(int x, int y, int z)
{
    return x + y + z;
}

int multiply(int x, int y)
{
    return x * y;
}

int main()
{
    printf("%d\n", multiply(add(1, 2, 3), 4));
    return 0;
}
```

Przykład D

```c
#include <stdio.h>

void doIt(int x)
{
    int y = 4;
    printf("doIt: x = %d y = %d\n", x, y);
    x = 3;
    printf("doIt: x = %d y = %d\n", x, y);
}

int main()
{
    int x = 1;
    int y = 2;

    printf("main: x = %d y = %d\n", x, y);

    doIt(x);

    printf("main: x = %d y = %d\n", x, y);

    return 0;
}
```

Zadanie 2.2

Napisz funkcję o nazwie `calculate()`, która przyjmuje dwie liczby całkowite oraz znak `char` reprezentujący jeden z następujących operatorów matematycznych: +, -, *, / lub % (reszta z dzielenia). Użyj instrukcji `switch`, aby wykonać odpowiednią operację matematyczną na liczbach całkowitych i zwróć wynik. Jeśli do funkcji zostanie przekazany nieprawidłowy operator, funkcja powinna wypisać komunikat o błędzie.

2 Przed przystąpieniem do zadań

Przed przystąpieniem do zadań należy:

1. Wejść w link z repozytorium GitHub znajdujący się na stronie kursu eNauczanie.
2. Zrobić fork repozytorium zgodnie z instrukcją do Laboratorium 1 (instrukcja na kursie).
3. Sklonować (pobrać) kod z własnego zforkowanego repozytorium za pomocą polecenia `git clone <skopiowany adres z GitHub>`.

W razie problemów proszę wrócić do instrukcji Laboratorium 1 lub zapraszam na konsultacje.

3 Zadania do wykonania podczas laboratorium

Podczas zajęć należy napisać prosty program "Kalkulator BMI".

Zakres programu:

1. Program pyta użytkownika o:
   - wzrost (w cm)
   - wagę (w kg)
2. Funkcja oblicza wskaźnik BMI.
3. Program działa w pętli do-while, aby użytkownik mógł wprowadzić kolejne dane bez ponownego uruchamiania programu.

Szkic kodu źródłowego:

```c
#include <stdio.h>
#include <stdbool.h>
// biblioteka w której jest funkcja potęgowania - pow() (opcjonalnie)
#include <math.h>

double calculateBMI(double weight, double height) {
    // oblicz BMI i zwróć wartość
}

int main() {
    double weight, height;
    char choice;

    do {
        // użytkownik wprowadza wagę i wzrost
        printf("Podaj wage (kg): ");
        scanf("%lf", &weight);

        printf("Podaj wzrost (cm): ");
        scanf("%lf", &height);

        // obliczanie BMI
        double bmi = calculateBMI(weight, height);
        printf("BMI = %.2f\n", bmi);

    } while (true);

    return 0;
}
```

Kolejnym zadaniem jest rozszerzenie programu o dodatkowe funkcjonalności:

1. Dodaj menu wyboru (switch) z następującymi opcjami:

- Oblicz BMI ($waga[kg]/(wzrost[m])^2$)
- Oblicz wskaźnik WHR (waist-to-hip ratio) $obwodTalii[cm]/obwodBioder[cm]$

2. Zaimplementuj funkcję double `calculateWHR`.
3. Uzupełnij program o odpowiednią walidację danych wejściowych (np. nie dopuszczaj wartości ujemnych).
4. W przypadku błędnego wyboru w menu – program powinien informować użytkownika i pozwolić na ponowną próbę.
5. Dodaj prostą pętlę for, która umożliwi obliczenie BMI dla kilku osób z rzędu (np. 3 osób) i wypisanie wyników.

4 Realizacja zadania wskazanego przez prowadzącego

Podczas zajęć prowadzący zleci realizację konkretnego zadania związanego z tematem laboratorium w języku C.