INFORMATYKA

TABLICE, PĘTLE I FUNKCJE W JĘZYKU C

CEL LABORATORIUM

Celem zajęć jest zapoznanie uczestniczki/uczestnika z tablicami, pętlami, funkcjami i wskaźnikami w języku C. Zajęcia kontynuują temat rozpoczęty poprzednio, jakim jest programowanie w języku C. Zaprezentowane zostaną tablice jedno i wielowymiarowe, pętle „*for*", „*while*" i „*do while*", funkcje i wskaźniki. Zajęcia zostaną zakończone serią zadań, łączących wszystkie poruszane tematy.

POCZĄTEK ZAJĘĆ

Uruchom program Oracle **VM VirtualBox** i maszynę wirtualną wskazaną przez osobę prowadzącą zajęcia. Jeżeli prace wykonujesz na maszynie wirtualnej SO_BD_IO, otwórz menu „`Start`", najedź kursorem na pozycję „`Wszystkie programy`", a następnie na „`Microsoft Visual Studio .NET 2003`". Możesz również skorzystać z własnego komputera lub z innych środowisk obecnych na komputerach w laboratorium. Upewnij się, że Twoje środowisko obsługuje języki C i C++.

Możesz skorzystać z projektu utworzonego na poprzednich zajęciach, jeżeli masz do niego dostęp. Możesz też założyć nowy projekt. Wybór należy do Ciebie.

TABLICE

Do tej pory zawsze tworzyliśmy zmienne, które przechowywały dokładnie jeden element. Jeżeli potrzebowaliśmy przechować trzy liczby, tworzyliśmy trzy osobne zmienne, żeby móc te liczby zapisać. Dość łatwo wymyślić sytuację, w której byłoby to rozwiązanie niepraktyczne. Co na przykład, jeżeli chcemy w naszym programie przeanalizować oceny z przedmiotu Informatyka studentów pierwszego roku Automatyki?

Takich ocen są dziesiątki albo nawet setki! Tworzenie osobnej zmiennej dla każdej oceny jest oczywiście możliwe, ale niesamowicie pracochłonne. Dennis Ritchie, twórca języka C, wyposażył język w specjalny rodzaj zmiennych, zwany tablicami. **Tablica** to zmienna, która pozwala przechować określoną liczbę wartości **takiego samego typu**.

Składnia instrukcji deklarowania i definiowania tablicy, przyjmuje następującą postać:

```
typ nazwa[rozmiar]; // deklaracja
typ nazwa[rozmiar] = {x0, x1, …, xrozmiar-1}; // definicja
```

Na przykład:

```
float oceny[1000];
char opcje[3] = {'a', 'b', 'c'};
```

Elementy tablic numerowane są począwszy **od zera**. To znaczy, że pierwszy element ma indeks 0, kolejny 1, kolejny 2, i tak dalej, aż do elementy o indeksie „rozmiar tablicy-1". Do poszczególnych elementów możemy się odwołać, podając nazwę tablicy i numer elementu w nawiasach kwadratowych, tak, jak pokazano w poniższym przykładzie:

```
#include <stdio.h>

int main(void)
{
	int fibonacci[5] = {0,1,1,2,3};

	printf("Pierwszy wyraz ciagu Fibonacciego ma wartosc: %i\n", fibonacci[0]);
	printf("Drugi wyraz ciagu Fibonacciego ma wartosc: %i\n", fibonacci[1]);
	printf("Trzeci wyraz ciagu Fibonacciego ma wartosc: %i\n", fibonacci[2]);
	printf("Czwarty wyraz ciagu Fibonacciego ma wartosc: %i\n", fibonacci[3]);
	printf("Piaty wyraz ciagu Fibonacciego ma wartosc: %i\n", fibonacci[4]);

	return 0;
}
```

Przepisz kod i zweryfikuj jego działanie.

ZADANIE 1

Zmień powyższy kod w taki sposób, żeby wypisał na ekranie „Kolejne elementy ciągu Fibonacciego mają wartość: …”. Zadanie zrealizuj za pomocą **dokładnie jednej** instrukcji `printf`.

TABLICE WIELOWYMIAROWE

Tablice mogą być wielowymiarowe. Tablica dwuwymiarowa, to w rzeczywistości *tablica tablic*. Tablica trójwymiarowa to *tablica tablic tablic* i tak dalej. Definicja takich tablic odbywa się poprzez zagnieżdżanie zbiorów wartości, a odwołanie do konkretnego elementu wymaga podania indeksu we **wszystkich** wymiarach. Poniższy kod prezentuje te elementy składni języka C:

```
#include <stdio.h>

int main(void)
{
	int tab1D[3] = {1,2,3};
	int tab2D[3][3] = { {1,2,3}, {4,5,6}, {7,8,9} };

	printf("%i", tab2D[1][1]); // Wypisze 5
	printf("%i", tab2D[1][2]); // Wypisze 6
	printf("%i", tab2D[2][0]); // Wypisze 7
}
```

PĘTLE

Oczywiście obsługa tablic poprzez odwoływanie się pojedynczo do każdego elementu, jest nadal niezmiernie praco i czasochłonna, co świetnie pokazuje poprzedni przykład. Chcieliśmy wypisać w nim 5 pierwszych liczb ciągu Fibonacciego. Co jednak w przypadku gdybyśmy liczb chcieli wypisać więcej (na przykład 100)?

Z pomocą przychodzą **pętle**. Z pętlami w ujęciu algorytmicznym, zapoznaliśmy się na drugich laboratoriach, przy okazji omawiania algorytmów. Tam, gdy potrzebowaliśmy „cofnąć się” w algorytmie do wcześniejszego kroku, używaliśmy instrukcji warunkowej, której jedna ze ścieżek, kierowała nas w tył, do fragmentu, który już wcześniej wykonano.

Pętla w języku C (i wielu innych językach) nie różnią się pod względem idei od znanego nam ujęcia algorytmicznego. Są trzy rodzaje pętli, jednak każda z nich zawiera warunek, którego spełnienie prowadzi

do powtórzenia oraz osobny blok kodu, który będzie powtarzany. Zacznijmy od najprostszej składniowo pętli, czyli pętli **while**. Jej składnia prezentuje się następująco:

```
while(warunekPowtarzania)
{
    // Powtarzany kod
}
```

Na przykład:

```
int a=0;
while(a<5)
{
    printf("%i", a);
    a++;
}
```

Powyższy kod wypisze liczby 0, 1, 2, 3 i 4. Liczba pięć **nie zostanie** wypisana, gdyż warunek (a<5), **nie będzie spełniony**, a warunek jest sprawdzany **przed** wykonaniem bloku pętli. Istnieje jednak pętla, której warunek jest sprawdzany **po** wykonaniu bloku (co znaczy, że blok pętli wykona się przynajmniej raz). Jest to pętla **do while**, której składnia wygląda w przedstawiony sposób:

```
do
{
    // Powtarzany kod
}
while(warunekPowtarzania)
```

Przykład rzeczywistego zastosowania:

```
int a=0;
do
{
    printf("%i", a);
    a++;
}
while(a<-1)
```

Powyższy kod wypisze liczbę 0, pomimo tego, że warunek powtarzania pętli, nie był nigdy spełniony (wartość zmiennej `a`, zawsze była większa od -1). Trzecim rodzajem pętli, jest pętla **for**. Składnia tej pętli jest bardziej skomplikowana, gdyż oprócz bloku kodu i warunku, zawiera też miejsce na **inicjalizację wartości zmiennych** i miejsce na zdefiniowanie **instrukcji krokowych**, wykonywanych w każdym kroku pętli, tuż po zakończeniu całego bloku kodu. Składnia tej pętli wygląda następująco:

```
for(inicjalizacja ; warunekPowtarzania ; instrukcjeKrokowe )
{
    // Powtarzany kod
}
```

Przykład rzeczywistego zastosowania:

```
int a = 10;
for(a=0; a<5; a++)
{
    printf("%i", a);
}
```

W miejscu inicjalizacji, wartość zmiennej `a` zostanie ustawiona na 0. Program wypisze więc liczby 0, 1, 2, 3 i 4, gdyż po każdym wykonaniu bloku pętli, wykonane zostaną instrukcje krokowe (w przykładzie jest tylko jedna instrukcja krokowa: `a++`). Liczba pięć **nie zostanie** wypisana, gdyż warunek (a<5), **nie będzie spełniony**, a warunek jest sprawdzany **przed** wykonaniem bloku pętli.

ZADANIE 2

Napisz program, który wypisze na ekranie komputera wszystkie liczby parzyste w przedziale od 1 do 160.

ZADANIE 3 Ⓟ

Napisz program, który będzie ciągle prosił użytkownika o podanie liczby zmiennoprzecinkowej. Jeżeli użytkownik wpisze 0, program wypisze na ekranie sumę liczb podanych przez użytkownika, zanim ten podał 0. Dla przykładu, jeżeli użytkownik poda 3, 4, -1, 0, program powinien wypisać „suma: 6” (gdyż 3+4-1=6).

FUNKCJE

Programy, które pisaliśmy, zawsze zawierały funkcję `main`. Do tej pory jednak, nie wyjaśniliśmy tego, dlaczego funkcja main wygląda właśnie w taki sposób. Wiemy natomiast, że w języku C, istnieją inne funkcje, gdyż omówiliśmy `printf` i `scanf`.

Okazuje się, że możliwe jest napisanie własnych funkcji, które spełniają określone zadania. Funkcje w języku C, są bardzo podobne do tych, które znasz z matematyki, jak chociażby funkcje trygonometryczne, logarytmy czy funkcja opisująca równanie kwadratowe. Każda z tych funkcji pobiera pewien argument (lub **argumenty**) i zwraca pewien **wynik**. Funkcje matematyczne pobierają i zwracają oczywiście liczby, ale język C pozwala na pobranie i zwrócenie dowolnego zadeklarowanego typu lub typów.

Deklaracja funkcji posiada stosunkowo prostą składnię. Najpierw określony jest typ zwracanej wartości, później nazwa funkcji (która musi spełniać takie same ograniczenia, jak nazwa zmiennej), a następnie jest zestaw okrągłych nawiasów, między którymi znajduje się **lista argumentów**. Lista argumentów zawiera dowolną liczbę par, z których każda opisuje jeden argument. Pierwszym elementem pary jest typ argumentu, a drugim nazwa. Jeżeli funkcja nie przyjmuje żadnych argumentów, nawias można zostawić pusty lub wpisać `void`. Tłumacząc opis na kod, dojdziemy do następującej postaci:

```
typZwracany nazwaFunkcji(typArg1 nazwaArg1, … , typArgN nazwaArgN);
```

Dla przykładu, moglibyśmy zadeklarować następującą funkcję:

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

Lub też taką:

```
int main(void);
```

Żeby funkcja działała, a program się kompilował, trzeba powiedzieć komputerowi, co funkcja będzie robić. Stąd, sama deklaracja nie wystarczy, konieczne jest napisanie definicji. Definicja funkcji różni się od deklaracji funkcji tym, że nie posiada średnika na końcu, ale posiada blok kodu (zwany „**ciałem funkcji**”), opisujący jej działanie:

```
typZwracany nazwaFunkcji(typArg1 nazwaArg1, … , typArgN nazwaArgN)
{
    // Kod funkcji
    // return wartoscTypuZwracanego, jezeli typZwracany inny niż void!!!
}
```

Przykładem są oczywiście wszystkie funkcje main, które do tej pory zdefiniowaliśmy. Wiemy też, jak wywołać funkcję, bo pracowaliśmy z funkcjami `printf` i `scanf`. Żeby funkcje wywołać, należy podać jej nazwę i wartości argumentów. Na przykład:

```
printf(„Hello world!”);
```

Albo:

```
float obliczonaWartosc;
float kat = 45.0;
obliczonaWartosc = sin(kat);
```

ZADANIE 4

Napisz funkcję obwodKola, która pobiera jeden argument typu zmiennoprzecinkowego i zwraca obwód koła o promieniu długości wartości podanego argumentu. Wartość liczby π możesz zaokrąglić do trzeciego miejsca po przecinku.

ZADANIE 5

W głównej funkcji programu wypisz obwód kół, o promieniach wyrażonych liczbami całkowitymi, większymi niż 1, a mniejszymi niż 30. Zastosuj pętlę `for` i funkcję napisaną w zadaniu 4.

DEKLARACJA, A DEFINICJA FUNKCJI

Deklarację funkcji można pominąć, jeżeli od razu ją zdefiniujemy. Często się tak robi, żeby zaoszczędzić czas. Kiedy deklaracja jest konieczna? Gdy potrzebujemy ją wywołać, zanim ją zdefiniujemy. Dzieje się tak chociażby wtedy, gdy dwie funkcje, w swoich ciałach, wywołują się wzajemnie. Niezależnie w jakiej kolejności je zdefiniujemy, zawsze któraś będzie pierwsza:

```
void a()
{
    b(); // Blad! ‘b’ nie zostalo zadeklarowane!
}
void b()
{
    a(); // To już w porzadku, definicja ‘a’ jest wyzej.
}
```

Najprostszym rozwiązaniem jest podanie deklaracji funkcji `b` przed definicją funkcji `a`:

```
void b();
void a()
{
    b(); // Wszystko w porzadku, kompilator wie, ze ‘b’ to funkcja.
}
void b()
{
    a(); // To tez w porzadku, definicja ‘a’ jest wyzej.
}
```

Należy także pamiętać, że definicja musi być zgodna z deklaracją. W szczególności tyczy się to typu zwracanego i listy argumentów.

KIEDY TWORZYĆ FUNKCJE

Na pewnym etapie programowania, będziesz zadawać sobie pytania „czy ten fragment kodu powinien być osobną funkcją?" i „co tak właściwie powinna ta funkcja robić?". Odpowiedzi możemy udzielić już teraz. Funkcją warto uczynić każdy fragment kodu, który będzie się powtarzać w takiej samej postaci, ale na innych wartościach wejściowych.

Jeżeli np. w naszym programie, mamy zamiar jednorazowo obliczyć obwód koła, warto to zrobić w kodzie funkcji main (lub gdziekolwiek indziej to czynimy) i nie wydzielać osobnej funkcji. Jeżeli jednak obwód koła będziemy musieli obliczyć dwa razy lub więcej, warto stworzyć osobną funkcję (np. `float obwodKola(float r)`).

Ważne jest, żeby pojedyncza funkcja miała jeden, konkretny cel, jasno go realizowała i nie robiła niczego, poza tym. Dla przykładu, w naszej funkcji `float obwodKola(float r)`, powinniśmy obwód zwrócić, a nie wypisać. Jeżeli chcemy obwód koła wypisać na ekranie, skorzystajmy z instrukcji `printf(„Obwód: %f", obwodKola(5));` lub zamiast tego, napiszmy funkcję `void wypiszObwodKola(float r)`.

WSKAŹNIKI

Ostatnim elementem języka C, który dziś omówimy, są **wskaźniki**. Wskaźnik to specjalny rodzaj zmiennej, który wyłącznie wskazuje na miejsce w pamięci, w którym jest przechowywana wartość określonego typu. Wskaźnik deklarujemy, umieszczając pomiędzy typem, a nazwą zmiennej, znak gwiazdki:

```
typ *nazwa;
```

Na przykład:

```
int *mojWskaznikNaTypInt;
```

Do wskaźnika, możemy przypisać wskaźnik tak, jak dowolną inną wartość, do dotychczasowych zmiennych. Jeżeli chcemy przypisać zmienną do wskaźnika, należy skorzystać ze znaku et („&"):

```
int *wskaznik1;
int *wskaznik2;
int zmienna = 10;

wskaznik1 = &zmienna; // Zmienna do wskaznika, wymagane &
wskaznik2 = wskaznik1; // Wskaznik do wskaznika, pomijamy &
```

Żeby wykonać odwrotną operację, to jest wartość zmiennej wskazywanej przez wskaźnik skopiować do innej zmiennej, należy natomiast zastosować znak gwiazdki:

```
zmienna = *wskaznik2;
```

Można pokusić się o stwierdzenie „skoro wskaźnik wyłącznie wskazuje, to znaczy, że nie przechowuje żadnej wartości, a więc nie jest zmienną". Zdanie to nie jest jednak prawdą. Każdy wskaźnik przechowuje **adres w pamięci**.

Co zapewne będzie zaskakujące dla Ciebie, ze wskaźnikami pracowaliśmy już wcześniej, już podczas poprzednich zajęć laboratoryjnych. Przypomnij sobie, że do funkcji scanf, parametry przekazywaliśmy w następujący sposób:

```
int zmienna;
scanf("%i", &zmienna);
```

A więc przypisywaliśmy zmienną do wskaźnika (tak jak w przykładzie wyżej: `wskaznik1 = &zmienna;`). Co więcej, wskaźnikami są również tablice. **Nazwa tablicy jest jednocześnie wskaźnikiem na jej pierwszy element**. Do każdego kolejnego elementu można się dostać poprzez dodawanie, jak w następującym przykładzie:

```
#include <stdio.h>
int main(void)
{
  int tablica[7] = {-4,18,1,33,-999,100,2};
  int i = 0;
  for(i=0;i<7;i++)
  {
        printf("Klasyczna metoda: %i\t", tablica[i]);
        printf("Metoda wskaznikowa: %i\n", *(tablica+i));
  }
  return 0;
}
```

Specjalnym rodzajem wskaźnika jest wskaźnik na typ `void`. Wskaźnikiem takim można wskazać na zmienną dowolnego typu

```
#include <stdio.h>

int main(void)
{
  void* dowolny;

  int a = 10;
  float b = 3.14;
  char c = 'x';

  dowolny = &a;
  dowolny = &b;
  dowolny = &c;

  return 0;
}
```

W tej chwili wskaźniki nie będą dla nas za bardzo przydatne, gdyż większość rzeczy możemy zrobić bez ich pomocy. Będą jednak potrzebne na kolejnych zajęciach. Warto więc już teraz się z nimi zaznajomić i przyzwyczaić się do korzystania z nich.

ZADANIA DO SAMODZIELNEGO WYKONANIA

ZADANIE 6

Przepisz następującą funkcję i przetestuj jej działanie na tablicy dziesięcioelementowej.

```
float minTablica(float *tablica, unsigned int rozmiarTablicy)
{
    float najmniejsza = tablica[0];
    int i;
    for(i=1; i<rozmiarTablicy; i++)
    {
        if(tablica[i]<najmniejsza)
        {
            najmniejsza = tablica[i];
        }
    }
    return najmniejsza;
}
```

ZADANIE 7

Zmień program z zadania 6 w taki sposób, aby zwracał największy element tablicy. Do elementów odwołuj się metodą wskaźnikową, a nie tablicową.

ZADANIE 8 Ⓟ

Napisz w języku C program, który definiuje trzy 10-elementowe tablice liczb całkowitych. Pierwsza z nich powinna zawierać liczby ze zbioru [-5,4], druga liczby ze zbioru [11,20], a trzecia liczby ze zbioru [-11,-20]. Zapytaj użytkownika o to, z której tablicy i którą wartość chce wypisać, a następnie wypisz ją na ekranie. Zadanie zrealizuj w taki sposób, żeby w całym programie były wyłącznie trzy instrukcje `printf`. Dwie pierwsze powinny zostać wykorzystane, żeby zapytać użytkownika o tablicę i liczbę.

ZADANIE 9 Ⓟ

Napisz w języku C program, który będzie prosił użytkownika o podanie znaku. Jeżeli znak będzie literą ze zbioru {a, g, k, m, n, t}, program powinien się zakończyć. W przeciwnym wypadku, powinien wyświetlić komunikat „Nieprawidlowy znak!” i poprosić użytkownika o ponowne podanie znaku.

ZADANIE 10 Ⓟ

Napisz w języku C funkcję, która pobiera pewną liczbę całkowitą N i zwraca N-tą liczbę Fibonacciego.

ZADANIE 11 Ⓟ (NAGRADZANE DWOMA PLUSAMI)

Napisz w języku C program, w którym zadeklarujesz tablicę o rozmiarach 10*10 elementów. Następnie napisz funkcję, której argumentem będzie tablica o takich wymiarach i która ustawi wartość każdego elementu na 2*x+y. Napisz drugą funkcję, która wypisuje tablice na ekranie, w formie macierzy. Elementy możesz odseparować znakiem specjalnym „\t”, czyli tabulatorem.

Autor:	Mgr inż. Paweł Stawarz, 15.10.2021
Korekta:	-