INFORMATYKA I: INSTRUKCJA 1

Wstęp

Rozpakuj projekt. Otwórz plik `projekt.sln`. W otwartym projekcie są następujące pliki:

1. `main.cpp` — główny plik z kodem. Tu piszemy nasz program
2. `winbgi2.cpp` — plik z funkcjami graficznymi
3. `winbgi2.h` — plik z definicjami funkcji graficznych

Pamiętaj: Często kompiluj projekt i patrz, czy wszystko działa!

1 Pierwsze kreski

Wewnątrz funkcji `main` wpisz:

```
graphics( 200, 200);
line( 0, 0, 200, 200);
line( 200, 200, 0, 0);
wait();
```

Uwaga: Zawsze pamiętaj o średnikach!
Skompiluj i uruchom projekt. Pierwsza linia tworzy okno grafiki, dwie następne rysują linie, zaś ostatnia czeka z zamknięciem okna na naciśnięcie dowolnego klawisza.

Ćwiczenia

Używając funkcji `line(x1,y1,x2,y2)` i `circle(x,y,r)`, wykonaj następujące zadania:

- Zidentyfikuj, jak ułożony jest układ współrzędnych (X,Y) w oknie.
- Narysuj kwadrat.
- Narysuj ludzika.
- Narysuj koła olimpijskie.

2 Zmienne

Pewne powtarzające się parametry (jak pozycję, promień, itp), możemy zastąpić zmiennymi. Następnie z nich wyliczyć odpowiednie współrzędne np:

```
int r,h;
h = 100;
r = 50;
line( 10, 0, 0, h);
line( 10, 0, 2*r, h);
circle( 10+r, h, r);
```

Możemy używać wszelkich działań i funkcji matematycznych: `+, -, *, /, sin(), ....`
Pamiętaj: Pierwsza linia deklaruje zmienne. Trzeba zadeklarować wszystkie zmienne, których będziesz używać! (szczegóły, na kolejnych zajęciach).
Zauważ, że wartość zmiennej jest nadpisywana, więc możemy napisać:

```
int w;
w = 50;
circle( 10, w, 10);
w = w + 20;
circle( 10, w, 10);
w = w + 20;
circle( 10, w, 10);
w = w + 20;
circle( 10, w, 10);
```

W efekcie wyświetlą się cztery kółka narysowane koło siebie. Przetestuj.

Ćwiczenia

Każdy program przetestuj dla paru ustawień zmiennych, by zobaczyć czy działa poprawnie.

- Napisz program, który dla zmiennych x,y,s, tworzy okno o rozmiarach x,y i na środku narysuje koło o promieniu s.
- Dla zmiennej `d`, narysuj cztery dotykające się koła o średnicy d w prawym górnym rogu okna.

- Dla zmiennej `y` narysuje koła olimpijskie w odległości `y` od górnej krawędzi.
- Skopiuj poprzedni kod trzy razy i w każdym fragmencie zmodyfikuj wartość zmiennej `y`.

3 Pętle

Pierwszą automatyzacją są pętle. Pętla wykonuje pewną operację, dopóki pewien warunek jest spełniony. Np:

```
int x;
x = 0;
while (x < 200) {
    line(x,10,x,190);
    x = x + 10;
}
```

Taki program będzie wykonywany w następujący sposób:

- wpisujemy 0 do zmiennej x
- sprawdzamy, czy $x < 200$
- rysujemy linię
- zwiększamy zmienną x o 10
- i znów: sprawdzamy, czy $x < 200$
- rysujemy linię
- zwiększamy zmienną x o 10
- sprawdzamy, czy $x < 200$
- rysujemy linię
- ...
- gdy wreszcie x przekroczy 200, pętla się skończy i program pójdzie dalej.

Ostatecznie program narysuje pionowe kreski dla kolejnych `x` $= 0, 10, 20, \ldots$.
Zauważ: Program nie narysuje linii dla $x = 200$, bo komputer najpierw sprawdzi, że $x \not< 200$ i przerwie pętlę.

Ćwiczenia

- Napisz program, który narysuje kratkę z odstępem `w`
- Narysuj rząd stycznych do siebie kół o promieniu r, zaczynając od lewej strony. Przemyśl: jeśli x to pozycja środka koła, to jaka wartość powinna jej być przypisana przed pętlą, o ile powinna być zwiększana i jaki warunek musi spełniać, by nie rysować poza oknem?!
- Pisząc jedną pętlę w drugiej, zapełnij cały obrazek przylegającymi kółkami.
- * Czy da się je lepiej upakować?
- Narysuj rząd kółek, których promienie zmniejszają się jak $\frac{1}{n}$.

Informatyka I: Laboratorium 2

4 Funkcje

Pewne zestawy operacji, zależne od zmiennych, możemy zebrać w grupki (funkcje) i wywoływać jak `circle` i `line`. Przykład z poprzedniego paragrafu możemy zamknąć w funkcji:

```
void obrazek(int h, int r)
{
    line( 10, 0, 0, h);
    line( 10, 0, 2*r, h);
    circle( 10+r, h, r);
}
```

Pierwsza linia deklaruje funkcję, która jest zależna od dwóch parametrów: `h,r`. Taką funkcję, możemy wywołać dla przykładu tak: `obrazek(100,50);`. Spowoduje to wykonanie powyższych trzech operacji przy $h = 100$ i $r = 50$.
Pamiętaj: Nową funkcję napisz przed funkcją `main`
W funkcji `main` wywołujemy funkcję `obrazek`, tak jak `circle` czy `line`:

```
void main()
{
    graphics(200,200);
    obrazek(100,50);
    wait();
}
```

Ćwiczenia

Napisz i wywołaj dowolne dwie z poniższych funkcji:

- `prostokat(x,y,a,b)` — Narysuje prostokąt o bokach `a` i `b` i środku w (`x`, `y`)
- `kwadrat(x,y,r)` — Narysuje kwadrat o boku $2r$ i wpisane koło o promieniu r.
- `ludzik(x,y,h)` — Narysuje ludzika wysokości `h` i środku głowy w (`x`, `y`)
- `olimpiada(x,y)` — Narysuje koła olimpijskie o środku w (`x`, `y`)
- * `okno(a)` — Używając funkcji do rysowania prostokąta narysuje okno o boku a.

5 Trochę więcej szczegółów

Omówmy pewne rzeczy trochę dokładniej.

5.1 Typy

W C i C++ musimy deklarować zmienne, tzn. powiedzieć, jakich będziemy używać zmiennych i jakich one będą typów. Deklaracje piszemy 'typ zmienna1,zmienna2, ...;'. Najważniejsze typy to:

- `int` — Liczba całkowita (32-bitowa, od -2^{31} do 2^{31})
- `float` — Liczba zmienno-przecinkowa. Może opisywać ułamki dziesiętne z ok. 7 cyframi znaczącymi (32-bity)
- `double` — Liczba zmienno-przecinkowa. Ma 16 cyfr znaczących (64-bity)

Pamiętaj: Jeśli używasz liczb rzeczywistych (a nie całkowitych), używaj typu `double`.
Pierwszym przykładem niech będzie:

```
double a;
a = 0;
while (a < 2*3.14)
{
    circle(a * 100, sin(a) * 100 + 100, 3);
    a = a + 0.001;
}
```

Ten program narysuje wykres sinusa przeskalowany o 100, za pomocą kółek o promieniu 3.

Ćwiczenia

Używając analogicznej pętli, wykonaj dowolne dwa z poniższych zadań.

- Narysuj wykres a^2.
- Narysuj punkty o współrzędnych $x = 100\sin a + 100$ i $y = 100\cos a + 100$.
- Narysuj punkty o współrzędnych $x = 100\sin a\cos 4a + 100$ i $y = 100\cos a\cos 4a + 100$.
- Narysuj punkty o współrzędnych $x = 100r\sin a + 100$ i $y = 100r\cos a + 100$, gdzie $r = \frac{\cos a+2}{3}$ (niech r będzie kolejną zmienną).

5.2 Typy — pułapki

Ważne, by pamiętać, że liczby bez przecinka dziesiętnego, są uważane za całkowite, tzn. wykonywane są na nich działania jak dla liczb całkowitych. Dlatego `1/4` da jako wynik 0! Bo wynik 0.25 zostanie obcięty do liczby całkowitej. Żeby tego uniknąć, możemy napisać `1.0/4` lub jeszcze lepiej `1.0/4.0`. Możemy także bezpośrednio 'zrzutować' zmienne z `int` na `double` pisząc: `(double) zmienna`.
Pamiętaj: Wszędzie, gdzie robisz obliczenia, używaj `double`. Unikaj mieszania liczb całkowitych i zmienno-przecinkowych. Nigdy nie pisz ułamków jako 1/3

Ćwiczenia

Przeanalizuj (i przetestuj) wynik tego programu. Które linie nie dadzą pożądanego efektu?

```
double a;
a = 0;
while (a < 2)
{
    circle(a * 100, sin( a * 3.14 ) * 100 + 100, 3);
    circle(a * 100, sin( a * (314 / 100) ) * 100 + 100, 3);
    circle(a * 100, sin( (a * 314) / 100 ) * 100 + 100, 3);
    a = a + 0.001;
}
```

6 Funkcje po raz drugi

Zestawy operacji, które powtarzamy w programie wielokrotnie, możemy zamknąć w funkcjach. Taka funkcja „połyka" parametry i coś z nimi robi. Dla przykładu:

```
void kreski(int n, double r)
{
    int i;
    i = 0;
    while (i < n)
    {
        line(i, 0, i, r * i);
        i = i + 1;
    }
}
```

W pierwszej linii mówimy:

- jak nazywa się funkcja — `kreski`
- jakie ma parametry — `n` typu `int` i `r` typu `double`
- jakiego typu zwraca wartość —w naszym wypadku `void` oznacza, że nic nie zwraca

Gdy gdziekolwiek w funkcji `main` użyjemy wywołania `kreski( 20, 0.4);` jako efekt działania funkcji otrzymamy 20 pionowych kresek o długości od 0 do $0.4 \cdot 19$ (dlaczego 19 a nie 20?).
Taką funkcję możemy wykonać wielokrotnie dla różnych wartości n i r:

```
void main()
{
    graphics(200,200);

    kreski( 10, 1.000);
    kreski( 20, 0.500);
    kreski( 30, 0.333);
    kreski( 40, 0.250);

    wait();
}
```

Ćwiczenia

Napisz i wywołaj dwie sposród niżej wymienionych funkcji.

- Funkcję, która narysuje ludzika wysokości `h` i środku głowy w (`x`, `y`).
- Funkcję, która w pętli narysuje tłum (używając poprzedniej funkcji).
- Funkcję, która narysuje `n` kółek w punkcie (`x`,`y`) o coraz większych promieniach.
- * Funkcję, która narysuje wielokąt foremny o `n` bokach.

7 Instrukcja warunkowa

Kolejnym bloczkiem składowym programowania, jest instrukcja warunkowa. Sprawdza ona warunek i wykonuje pewną część kodu, tylko gdy warunek jest spełniony.

```
x = 2.0;
if ( x > 0 ) {
    y = sqrt(x);
} else {
    y = 0;
}
```

Instrukcja ta sprawdza czy $x > 0$ i jeśli jest to prawdą, to wstawia $\sqrt{x}$ do zmiennej y. Gdy warunek nie jest spełniony, wykonywana jest część po `else`, więc wstawiane jest 0 do y. W ten sposób możemy zabezpieczyć się na przykład przed niemożliwymi obliczeniami, albo uzależnić działanie programu od jakiś wartości.
Zobaczmy prosty przykład:

```
double a;
a = 0;
while (a < 2*3.14)
{
    if (a < 2) {
        circle(sin(a) * 100 + 100, cos(a) * 100 + 100, 5);
    } else {
        circle(sin(a) * 100 + 100, cos(a) * 100 + 100, 10);
    }
```

```
    a = a + 0.001;
}
```

Gdyby nie instrukcja `if`, ten program narysował by koło z małych kółek. Teraz, gdy kąt `a` przekroczy 2 radiany zmieni promień kółeczka z 5 na 10

Ćwiczenia

Napisz program który:

- Dla parametru w rysuje wykres $x^2 - w$, przeskalowany o 100 w obu kierunkach i przesunięty na środek (patrz poprzedni przykład).
- Wyrysuje większe kółka w miejscach przecięcia wykresu z osią x (jeżeli przecina).
- Zmodyfikuj program by działał dla dowolnych a, b, c i funkcji ax^2+bx+c.

Informatyka I: Instrukcja 3

Informatyka I: Instrukcja 3

8 Instrukcje wejścia/wyjścia

Praktyczny program powinien mieć możliwość interaktywnej komunikacji z użytkownikiem. Do drukowania informacji dla użytkownika służy najczęściej standardowe wyjście (monitor). W nowym projekcie pakietu MS Visual Studio (poproś prowadzącego, aby pokazał, jak stworzyć **pusty** projekt), napisz program, który wydrukuje tekst *Witaj na trzecim laboratorium!*

```
void main()
{
   printf("Witaj na trzecim laboratorium!");
}
```

Instrukcja `printf` służy do wypisywania tekstu na ekran. Jako argument przyjmuje zmienną typu tekstowego. Do formatowania tekstu służą *sekwencje formatujące*, które pozwalają wprowadzić znak nowej linii, tabulacji itp. Umieszczona wewnątrz tekstu sekwencja znaków:

- \n — wprowadza znak nowej linii,
- \t — wprowadzan znak tabulacji.

Ćwiczenia

Używając **jednej** instrukcji `printf` oraz odpowiednich sekwencji formatujących, wygeneruj tekst identyczny z poniższym:

```
To jest pierwsze zdanie w mojej instrukcji.
To jest tuz po znaku nowej linii.      Zas ten fragment
                                       oddzielony jest znakiem
                                       tabulacji!
Za to w ponizszej linii wszystkie liczby oddzielono tabulatorami.
5.2    3.14    -7    8
```

Uwaga

Oczywiście wprowadzenie długiego tekstu (np. kilku komunikatów dla użytkownika) w jednej instrukcji `printf` jest nonsensem. Spróbuj osiągnąć ten sam efekt, co powyżej, ale tym razem użyj osobnej instrukcji `printf` dla każdego ze zdań. Czy coś cię zaskakuje? Czy nowa instrukcja `printf` wymusza przejście do nowej linii?
W instrukcji `printf` nie używaj polskich znaków diakrytycznych. Da się to zrobić, jednak wymaga pewnych komplikacji i w prostych programach nie jest praktykowane. Jeśli bardzo cię męczy ciekawość, w wolnej chwili poszukaj rozwiązań w książkach, bądź internecie.

Dalej o `printf`

Pewne znaki specjalne są w języku C zarezerwowane na potrzeby konkretnych instrukcji. Wiele z nich poznasz wkrótce. Dobrymi przykładami takich znaków są % czy backslash \. Nie mogą one być użyte wprost, gdyż mają swoje funkcje w języku C. Jeśli chcesz, by się pojawiły na ekranie, musisz poprzedzić je dodatkowym znakiem \.

- Dopisz do swojego programu intstrukcję, która wydrukuje następujący tekst:

  ```
  82% dysku C:\ jest w uzyciu!
  ```

Program o znaczeniu inżynierskim musi jednak mieć możliwość drukowania na ekran liczb i wyników przeprowadzonych działań.

- Przepisz do funkcji main następujące instrukcje:

```
int a = 5;
double c = 8.2;

printf("Zmienna a ma wartosc %d, zas zmienna c = %lf\n", a, c);

c = c + 7.5;
c -= a;
a = 1;
c -= 2*a;

printf("Po dodaniu do zmienej c wartosci 7.5, odjeciu a
        oraz odjeciu dwukrotnosci zmodyfikowanej
        wartosci a zmienna c = %lf\n", c);
```

- Przeanalizuj dokładnie kod. Pojawiają się w nim nowe instrukcje arytmetyczne!
- Między wszystkimi instrukcjami arytmetycznymi dodaj po jednej linijce kodu, który wydrukuje na ekran bieżącą wartość przechowywaną w zmiennych a i c.

Pojawiły się też nowe elementy. Do drukowania wartości przechowywanych w zmiennych służą *sekwencje formatujące* lub inaczej *specyfikatory formatu*. Są one następujące:

- %lf — dla zmiennych typu `double`
- %d — dla zmiennych typu `int`
- %f — dla zmiennych typu `float`

Dodatkowo, dla liczb zmiennoprzecinkowych o ekstremalnie małych, umiarkowanych i ogromnych wartościach użyj poniższych sekwencji i zobacz, jaki będzie efekt działania.

- %lg, %e, %.2lf, %.4lf (dla zmiennych typu `double`),
- %.3f (dla zmiennych typu `float`).

Czytanie z klawiatury

Instrukcją służącą do czytania danych ze standardowego wejścia (klawiatury) jest instrukcja `scanf`. Przykłady jej użycia wyglądają następująco:

```
int a;
scanf("%d", &a);

double c;
scanf("%lf", &c);

int b, d;
double g, h;
scanf("%lf%d%d%lf", &g, &d, &b, &h);
```

Uwaga: Zwróć szczególną uwagę na znak & występujący przed nazwami zmiennych, do których wczytujemy wartości. Znak ten **nigdy** nie występuje w intrukcji `printf`, za to zawsze jest potrzebny w instrukcji `scanf`.
Zauważ również, że używając jednej instrukcji `scanf` możesz wczytać wiele liczb. Sekwencje formatujące nie muszą być oddzielone spacjami, za to wartości muszą być podane z klawiatury w odpowiedniej kolejności - takiej, w jakiej zmienne na liście argumentów, do których te wartości mają trafić.

Ćwiczenia

Napisz prosty kalkulator, który wczyta z klawiatury dwie liczby typu rzeczywistego i wykona na nich dodawanie, odejmowanie, mnożenie i dzielenie. Odejmowanie i dzielenie oczywiście nie jest przemienne. Policz zatem każdą z możliwych różnic czy ilorazów. Wydrukuj wszystkie wyniki na ekran.

9 Jeszcze trochę o funkcjach

Funkcje nie tylko grupują pewne logicznie wydzielone bloki instrukcji, których używamy wielokrotnie (jak funkcja rysująca ludzika z kółek i kresek, bądź funkcja rysująca tłum z użyciem funkcji `ludzik`). Do tej pory ich deklaracje i definicje wyglądały odpowiednio tak:

```
void NazwaFunkcji(int argument1, double argument2);

void NazwaFunkcji(int argument1, double argument2)
{
    // Tu sie znajduje cialo funkcji
}
```

Funkcje mogą bowiem zwracać wartość. Typ zmiennej, jaką zwracają jest zawsze identyczny z typem funkcji. Nie musi za to być zgodny z typami argumentów, których typy mogą być zupełnie inne. Weźmy dla przykładu funkcję, która przyjmie dwie wartości (jedną typu `double`, drugą typu `float`) i zwróci liczbę całkowitą równą 5, gdy większą wartość ma pierwszy argument lub wartość 10 w przeciwnym razie. Przeanalizujmy odpowiednio deklarację i kod takiej funkcji.

```
int KtoryWiekszy(double a, float b);

int KtoryWiekszy(double a, float b)
{
   int Wynik;

   if(a > b)
   {
      Wynik = 5;
   }
   else
   {
      Wynik = 10;
   }
   return Wynik;
}
```

Zwróć uwagę na instrukcję `return`, która zwraca z funkcji **wartość przechowywaną w konkretnej zmiennej**. To ważne! Funkcja nigdy nie zwraca zmiennej. Zwraca tylko wartość, jaka była w tej zmiennej przechowywana. Ponadto zmienna zadeklarowana w danej funkcji będzie dla programu widoczna **tylko i wyłącznie wewnątrz tej funkcji**, a nie będzie rozpoznawana w innych fragmentach kodu (np. funkcji `main`). Prześledźmy jeszcze kod funkcji `main`, w której występuje wywołanie naszej funkcji.

```
void main()
{
   float c = 8.14;
   double d = -7.3814;
   int InnaZmienna = 15;

   KtoryWiekszy(d, c);
   InnaZmienna = KtoryWiekszy(d, c);
   InnaZmienna = KtoryWiekszy(12.5, c);
}
```

Dodaj do powyższego kodu instrukcje, które po każdym wywołaniu funkcji `KtoryWiekszy` wydrukują wartość aktualnie przechowywaną w zmiennej `InnaZmienna`. Zastanów się, jaki będzie wynik i sprawdź, czy masz rację.
Zmodyfikuj napisany dziś kalkulator tak, aby instrukcje sumowania, odejmowania, mnożenia i dzielenia były realizowane przez osobne funkcje `Sumuj`, `Odejmij`, `Pomnoz`, `Podziel`. Funkcje te musisz napisać samodzielnie.

10 *Coś na deser

Drukowanie tekstów na ekran nie musi sprowadzać się tylko do drukowania napisów, które są na twardo zdefiniowane w kodzie źródłowym lub wartości przechowywanych w zmiennych liczbowych. Język C ma również odpowiedni typ na przechowywanie zmiennych tekstowych, których zawartość może dynamicznie się zmieniać w trakcie wykonywania programu. Spróbuj zrozumieć i skompilować poniższy kod. Więcej szczegółów stanie się dla Ciebie jasnych, gdy omówione zostaną *tablice*.

```
void main()
{
   char tekst[] = "To jest moj tekst\n";

   printf(tekst);
}
```

Istnieje również szereg funkcji, które pozwalają łączyć teksty, porównywać je ze sobą, przekształcać zmienne liczbowe do postaci zmiennych tekstowych. Zainteresowanych odsyłamy do zewnętrznych materiałów poświęconych *zmiennym łańcuchowym (ang. string)*.

Informatyka I: Instrukcja 4

Liczby losowe

Generacja liczb losowych jest bardzo przydatna w wielu obszarach począwszy od komputerowych gier w kości a na skomplikowanych symulacjach mechanicznych i kryptografii skończywszy. Poniższe zadania przedstawiają mechanizm

generacji liczb pseudolosowych w języku C/C++[1].

Generatory liczb pseudolosowych wymagają zastosowania tzw. ziarna czyli liczby, która posłuży do inicjalizacji procesu losowania kolejnych liczb. Zazwyczaj do inicjalizacji generatora używa się czasu sczytywanego z procesora komputera, więc konieczne będzie dołączenie biblioteki `time.h`
Poniższy program generuje losową liczbę całkowitą z przedziału 0 do RAND_MAX. RAND_MAX jest zdefiniowane w bibliotece `stdlib.h`, jest ono postaci 2^{n-1} np. 32767.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

void main()
{
     int i;

     srand ( time(NULL) );        //inicjalizacja generatora
     i = rand();                  //funkcja losująca liczbę
     printf(" Wylosowana liczba to %d\n", i);
}
```

Ćwiczenia

1. Wykonaj powyższy program kilka razy z rzędu. Co możesz powiedzieć o wyniku?
2. Zastąp ziarno generatora `time(NULL)` stałą liczbą. Co się wówczas stanie?
3. Zmodyfikuj program tak, by losował ciąg 40 liczb psedudolosowych i wypisywał je na ekran.
4. Wypisz na ekran wartość RAND_MAX.

Zazwyczaj interesuje nas konkretny przedział liczb. Aby określić kres dolny oraz długość takiego przedziału musimy dokonać kilku transformacji na wylosowanych liczbach:

[1]Mówimy, że generowane liczby są pseudolosowe, gdyż niemożliwe jest wygenerowanie prawdziwie losowego ciągu liczb. W praktyce świetnym źródłem ciągów liczb losowych są dane uzyskiwane z natury np. dane meteorologiczne (www.random.org).

Po prostu bierzemy wyłącznie resztę z dzielenia przez zadaną długość przedziału i dodajemy kres dolny jak w poniższym fragmencie kodu:

```
//  kod losuje liczby z przedzialu [ 20 do 50 ]

int min = 20;
int max = 50;
int L = max - min + 1;

i = rand()%L + min;
```

operator % liczy resztę z dzielenia jednej liczby przez drugą. Tutaj, licząc resztę z dzielenia wyniku losowania przez 31, otrzymujemy liczbę z przedziału 0 do 30.

Uwaga

Czy taki wzór będzie generował liczby losowe o rokładzie równomiernym? W domu zastanów się nad lepszym rozwiązaniem, a poniżej wykorzystaj to najprostsze.

Ćwiczenia

1. Zmodyfikuj powyższy program tak, aby generował liczby z przedziału od 0 do 100.
2. Wybierz krótki przedział (rzędu kilku możliwych do wylosowania liczb) i wykonaj 1000 oraz 10000 losowań. W obu przypadkach zliczaj, ile razy wylosowano każdą liczbę.
3. Wynik (informację o tym, ile razy wylosowano każdą z wartości) wydrukuj na ekranie. Co możesz powiedzieć o rozkładzie tego losowania?

Liczby losowe typu rzeczywistego, rzutowanie

Do tej pory losowane liczby były liczbami całkowitymi. Często potrzebujemy liczb zmiennoprzecinkowych. W tym celu po prostu przeskalujemy wylosowaną liczbę tak, aby zawsze należała do przedziału 0-1. Wystarczy podzielić wylosowaną liczbę przez RAND_MAX. Dodatkowo należy pamiętać, że wylosowana liczba jest typu całkowitego `int`, więc dzielenie przez RAND_MAX da

nam zazwyczaj 0. Aby tego uniknąć, musimy przekonwertować typ `int` na typ zmiennoprzecinkowy `double`. Taką operację nazywamy rzutowaniem. Rzutowanie ma bardzo szerokie zastosowanie i odnosi się nie tylko do operacji na liczbach. Poniższy kod losuje liczbę z przedziału 0-1:

```
double x;
srand(  time  (NULL)  );

// instrukcja (double) rzutuje typ int na double
x = (double)rand()/RAND_MAX;
```

Ćwiczenia

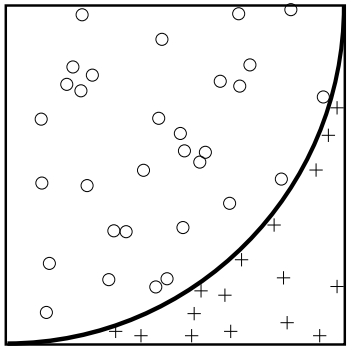
1. Napisz funkcję `isInCircle` typu `int`, która będzie losowała punkt w kwadracie $[0,1]\times[0,1]$ i zwracała wartość 1, jeśli punkt znajduje się wewnątrz koła o promieniu 1. Jeśli punkt jest poza kołem, niech funkcja zwraca 0.

2. W programie głównym sprawdź, czy punkt wylosowany przez daną funkcję jest w obszarze koła, czy nie. Zauważ, że nie masz dostępu do współrzędnych wylosowanych przez funkcję. Funkcja zwraca tylko wartość logiczną.

3. Niech powyższa funkcja dodatkowo zaznacza dany punkt na ekranie. Jeśli jest on wewnątrz koła, niech oznacza go kółkiem. Jeśli jest na zewnątrz, niech oznacza go krzyżykiem. Najwygodniej będzie przeskalować dane zaznaczenie 200-krotnie. Dodatkowo, narysuj na ekranie ćwiartkę koła o promieniu 200 oraz kwadrat $[0,200]\times[0,200]$, aby było widać, że wylosowany punkt rzeczysićie jest wewnatrz koła.

4. Wywołaj powyższą funkcję 100-krotnie, aby sprawdzić, jak działa[2]. Wynik działania powinien przypominać rysunek poniżej.

[2]Funkcję zwracajacą typ możemy wywołać zarówno przypisując wartość, która jest przez nią zwracana do jakiejś zmiennej np. `a = isInCircle()` jak i po prostu wywołać ją w programie bez przypisania, pisząc po prostu `isInCircle()`. Wówczas wartość zwracana przez funkcję przepadnie, ale wszystkie inne rzeczy, które się dzieją w funkcji (u nas jest to rysowanie), zostaną i tak wykonane



Rysunek 1: Ćwiartka koła w kwadracie.

Liczenie przybliżeń liczby π metodą Monte Carlo

Koło wpisane w jednostkowy kwadrat ma powierzchnię równą $\pi/4$. Toteż losując punkty w kwadracie z prawdopodobieństwem $\pi/4$ trafiamy w obszar koła. Dzięki temu, zliczając odsetek wylosowanych punktów, które trafiły do wnętrza koła, możemy określić przybliżenie liczby π. Taka metoda zalicza się do metod Monte Carlo[3].

Ćwiczenia

1. Przy użyciu poprzedniej funkcji policz przybliżenie liczby π. Wykonaj 10000 losowań.

2. Sprawdź, jak zmienia się dokładność przybliżeń, gdy wykonujemy coraz więcej losowań. Wykonaj 100, 1000 i 100000 losowań.

3. * Wydrukuj na ekran zależność względnego błędu przybliżenia π od potęgi liczby 2 w zakresie 2^8 do 2^{32}.

Informatyka I: Instrukcja 4+

[3]w Monte Carlo mieści się bodaj najsłynniejsze w Europie kasyno, stąd taka nazwa dla metod losowych

11 Wskaźniki i referencje - bezboleśnie

Nauczyliśmy się do tej pory, że funkcje w języku C mogą zwracać wartość. Co jednak, gdybyśmy chcieli napisać funkcję, która rozwiąże równanie kwadratowe i zwróci jego dwa rozwiązania? Jest to niemożliwe z użyciem tego mechanizmu - funkcja może zwrócić bowiem jedną i tylko jedną wartość! Do zwrócenia dwóch wyników potrzebujemy innego mechanizmu. Nie możemy tego zrobić z użyciem wartości zwracanej przez funkcję. Wyobraźmy sobie następujący kod funkcji `main`.

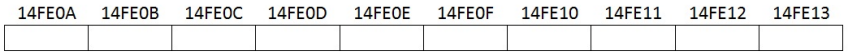
```
void main()
{
   double a, b, c, x1, x2;
   a = 1;
   b = -3;
   c = 2;

   RozwiazRownanieKwadratowe(a, b, c, ...inne argumenty...);

   printf("x1 = %lf, x2 = %lf\n", x1, x2);
}
```

Powyżej zależałoby nam na tym, aby funkcja `RozwiazRownanieKwadratowe` była w stanie „w jakiś sposób" wpisać rozwiązanie do zmiennych `a` i `b` zadeklarowanych wewnątrz funkcji `main()`. Taki mechanizm zapewniają w języku C wskaźniki.

12 Wskaźniki - trochę istotnej teorii

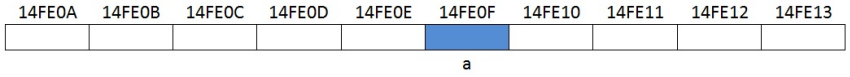
W pewnym uproszczeniu pamięć operacyjną komputera (w której przechowywana jest każda ze zmiennych, jakie deklarujemy w kodzie - de facto wartość tej zmiennej) można postrzegać jako zbiór komórek. To tak zwany liniowy model pamięci. Każda z tych komórek może przechowywać jedną wartość. Każda ma też swój adres (komputer musi się do nich jakoś odnosić). Adresy przedstawione są jako liczby w formacie szesnastkowym (patrz Rys. 1). Mając na uwadze taki model pamięci, prześledźmy następujący kod.

```
void main()
{
   int a;
}
```



Rysunek 2: Liniowy model pamięci

W tym momencie dokonaliśmy deklaracji zmiennej `a`, tzn. zarezerwowana została dla niej jakaś komórka w pamięci komputera. Podczas pisania tej instrukcji, mój komputer zarezerwował dla zmiennej `a` komórkę o adresie 14FE0F. Obecny stan obrazuje Rys. 2. Zacienienie oznacza, że zadeklarowaliśmy zmienną i to miejsce zostało zarezerwowane właśnie dla niej. Pod komórką (w celach czysto dydaktycznych) podpisaliśmy, jaka zmienna jest przechowywana w danej komórce pamięci.

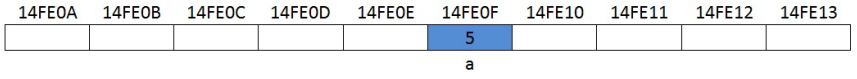


Rysunek 3: Zadeklarowana zmienna w pamięci

Teraz dokonamy przypisania wartości do zmiennej `a`.

```
void main()
{
   int a;
   a = 5;
}
```

Obecną sytuację obrazuje Rys. 3.



Rysunek 4: Zadeklarowana zmienna w pamięci oraz przypisana do niej wartość

Wydrukujmy teraz na ekran adres komórki, w jakiej przechowywana jest zmienna `a` oraz samą wartość tam przechowywaną (czyli de facto wartość zmiennej `a`). Operatorem wyłuskania adresu jest operator `&`. Aby więc uzyskać adres zmiennej `a`, trzeba napisać `&a`. Przepisz poniższy kod, który ilustruje powyższy opis.

```
void main()
{
   int a;
   a = 5;
   printf("Adres komorki to %X, a wartosc to %d.\n", &a, a);
}
```

Powyższa sekwencja formatująca `%X` nie różni się niczym od standardowego `%d` poza tym, że wydrukuje liczbę w notacji szesnastkowej, a nie dziesiętnej. Powinieneś na ekranie zobaczyć wynik zbliżony do poniższego.

```
Adres komorki to 14FE0F, a wartosc to 5.
```

Dalej o wskaźnikach i referencjach

Wróćmy teraz do naszego przykładu funkcji obliczającej pierwiastki równania kwadratowego i mającej „w jakiś sposób" zwrócić do funkcji `main` dwa rozwiązania. Wiemy, że wykorzystywana przez nas wcześniej instrukcja `return` zwraca wartość jako wynik działania funkcji. To ważne! Zwraca wartość, a nie jakąś zmienną. To tak (upraszczając), jakby w jakimś biurze siedziała pani wykonująca te same zadania, co nasza funkcja, a wynik (wartość) zwracała nam jako liczbę zapisaną na kartce. Oczywiście dostajemy więc samą liczbę i nie wiemy nic o tym, co ta pani policzyła (lub jakich wzorów użyła i jakimi literkami sobie oznaczyła poszczególne wielkości). My (w funkcji `main`) następnie bierzemy tę kartkę i zapisaną na niej liczbę wpisujemy w jakieś miejsce w pamięci. Tyle w funkcji `main` oznacza taki kawałek kodu:

```
c = Suma(a, b);
```

Chcemy jednak zwrócić dwie wartości. W powyższy sposób nie da się tego zrealizować w języku C (pani nie zna żadnego mechanizmu wydania nam dwóch kartek). Ale gdyby udało nam się powiedzieć tej pani, że gdy już się w swoim biurze doliczy dwóch wyników, to zamiast cokolwiek zapisywać na kartce (i oddawać nam) ma po prostu kartkę z jednym wynikiem zanieść do domu stojącego pod jednym adresem (czyli wpisać jakąś wartość do zmiennej w pamięci komputera), a kartkę z drugim wynikiem zanieść do domu stojącego pod drugim adresem, to po zakończeniu działania funkcji (mimo, że funkcja nie zwróciła przez wartość zwracaną żadnego wyniku) mielibyśmy oba wyniki wpisane w odpowiednich miejscach w pamięci. Przejdźmy do formalizmu języka C. Napiszmy funkcję `DodajOdejmij`, która przyjmie dwie wartości typu `double`, obliczy sumę i różnicę tych liczb i nie zwróci żadnego wyniku przez wartość zwracaną, ale wpisze zarówno sumę, jak i różnicę bezpośrednio w odpowiednie miejsca w pamięci. Zgodnie z powyższą historyjką, musi w tym celu znać też adresy tych zmiennych. Przyjmie je również na liście argumentów. Przepisz poniższy kod funkcji.

```
void DodajOdejmij(double a, double b, double &suma, double &roznica)
{
   suma = a + b;
   roznica = a - b;
}
```

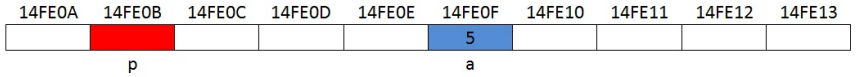
Na liście argumentów poinformaliśmy znakiem `&` kompilator o tym, że funkcja przyjmie adresy, a wewnątrz ciała funkcji możemy już używać zmiennych `suma` i `roznica` jako normalnych zmiennych. Dopiszmy jeszcze kod funkcji `main`.

```
void main()
{
   double a = 12;
   double b = 10;
   double suma, roznica;

   DodajOdejmij(a, b, suma, roznica);

   printf("Suma = %lf, Roznica = %lf\n", suma, roznica);
}
```

Właśnie zwróciliśmy z funkcji dwie wartości przez referencję (czyli tak naprawdę adres). Przyjrzyjmy się jeszcze wskaźnikom. Wróćmy w tym celu do naszego liniowego modelu pamięci komputera. Wskaźnik to taka zmienna, która wskazuje miejsce w pamięci, w którym przechowywana jest inna zmienna. Możemy na przykład pokazywać na zmienną `a`. Wskaźnik (zmienną typu wskaźnikowego) deklarujemy przez poprzedzenie jej nazwy znakiem `*`. Przyjrzyjmy się przykładowi.

```
void main()
{
   int a;
   a = 5;
   int *p;
}
```

W tym momencie a jest zadeklarowane jak poprzednio, zaś p to zmienna, która ma na coś pokazywać. Niemniej to też zwykła zmienna, więc zarezerwowano dla niej jakieś miejsce w pamięci (Rys. 4).

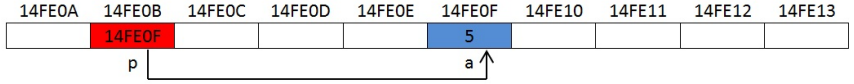


Rysunek 5: Zadeklarowana zmienna p oraz a (na razie nic o sobie nie wiedzą)

Zmienna p jednak na razie na nic nie pokazuje. Jest dokładnie taką samą zmienną, jak każda inna. Trzeba jej więc przypisać, na co ma pokazywać. Chcemy, aby pokazywała na zmienną a. Pokazywać to znaczy nic innego niż znać adres, gdzie przechowywana jest zmienna a. Wtedy każdemu, komu wskaźnik p ma wskazać zmienną a mówi po prostu, gdzie ona „mieszka". Dokonajmy więc tego przypisania. p musi przechowywać adres zmiennej a.

```
void main()
{
   int a;
   a = 5;
   int *p;
   p = &a;
}
```

Dopiero teraz stworzyliśmy powiązanie! p pokazuje na a (Rys. 5).



Rysunek 6: Zadeklarowana zmienna p (wskaźnik), który teraz już pokazuje na zmienną a (zna po prostu jej adres - tzn. wartość przechowywana w zmiennej p to adres zmiennej a)

Zapamiętaj!

Zmienna p przechowuje adres a. Natomiast używając zapisu *p możesz wyłuskać wartość, na jaką p pokazuje. Czyli my wyłuskamy wartość przechowywaną w zmiennej a.
Przepisz teraz kod, który zobrazuje to, czego dokonaliśmy.

```
void main()
{
   int a;
   a = 5;
   int *p;
   p = &a;

   printf("Wartosc a = %d, adres a = %X\n", a, &a);
   printf("Wartosc p = %X, wartosc pokazywana przez p = %d\n", p, *p);
}
```

Skompiluj i uruchom program. Powinienes uzyskać napis zbliżony do poniższego.

```
Wartosc a = 5, adres a = 14FE0F
Wartosc p = 14FE0F, wartosc pokazywana przez p = 5
```

Przepisz jeszcze poniższy program i samodzielnie (wstawiając w odpowiednich miejscach instrukcje printf drukujące wartość zmiennej a) przeanalizuj, jak działa.

```
void main()
{
   int a;
   int *p;
   p = &a;

   *p = 8; // wpisujemy 8 w miejsce pokazywane przez p
         // Jaka jest teraz wartosc a?

   a = 10; // Jaka jest teraz wartosc wskazywana przez p?
}
```

13 Zwracanie wartości przez wskaźnik

Wcześniej napisaliśmy funkcję, która zwraca dwie wartości przez referencję. Można tego samego dokonać przez wskaźnik. Analogiczny kod funkcji DodajOdejmij ma następującą składnię (teraz zamiast referencji na liście argumentów przekazujemy wskaźniki).

```
void DodajOdejmij(double a, double b, double *suma, double *roznica)
```

```
{
   *suma = a + b;
   *roznica = a - b;
}
```

Pamiętaj, że w przypadku użycia wskaźników wewnątrz funkcji musisz używać cały czas operatora `*`, aby przypisać coś do wartości wskazywanej przez wskaźnik. Kod funkcji `main` jest teraz taki:

```
void main()
{
   double a = 12;
   double b = 10;
   double suma, roznica;

   // Deklarujemy wskazniki
   double *WskaznikNaSuma, *WskaznikNaRoznica;

   // Dokonujemy przypisania do zmiennych, na ktore maja pokazywac
   WskaznikNaSuma = &suma;
   WskaznikNaRoznica = &roznica;

   DodajOdejmij(a, b, WskaznikNaSuma, WskaznikNaRoznica);

   printf("Suma = %lf, Roznica = %lf\n", suma, roznica);
}
```

Zauważmy jeszcze na koniec, że jawne deklarowanie wskaźników na sumę i różnicę oraz jawne powiązywanie ich z sumą i różnicą jest zbędne. Możemy tego dokonać wprost przy wywołaniu funkcji (na liście jej argumentów), bowiem tylko do funkcji muszą trafić odpowiednio powiązane wskaźniki, a de facto adresy (bo właśnie adresy wskaźniki przechowują). Gdy dokonamy tego uproszczenia, funkcja `main` ma następującą formę. Przeanalizuj bardzo dokładnie, na czym polega uproszczenie.

```
void main()
{
   double a = 12;
   double b = 10;
   double suma, roznica;

   DodajOdejmij(a, b, &suma, &roznica);

   printf("Suma = %lf, Roznica = %lf\n", suma, roznica);
}
```

Ćwiczenia

Napisz funkcję, która rozwiąże równanie kwadratowe (przy danych współczynnikach `a, b, c`) i zwróci oba rozwiązania. Napisz dwa warianty tej funkcji - jeden, który dokona tego przez referencję; drugi - który dokona tego przez wskaźniki. Odpowiednio dostosuj funkcję `main` do każdego z wariantów.

Zapamiętaj!

Do poprzednich zajęć wszystkie nasze funkcje przyjmowały argumenty jako wartości (tworzone były dla nich lokalne kopie samych wartości przechowywanych w zmiennych) i również zwracały wartości (a nie zmienne). Tym samym funkcje mogły zmieniać swoje lokalne kopie pewnych zmiennych, ale nie wpływało to na wartości tych zmiennych poza tą funkcją. Natomiast za każdym razem, gdy używasz wskaźnika, bądź referencji, działasz na rzeczywistym miejscu w pamięci, w którym ta zmienna jest przechowywana. Tzn. że tak dokonane modyfikacje będą widziane na zewnątrz funkcji po jej wykonaniu. Obrazuje to poniższy przykład. Po każdym wywołaniu dowolnej funkcji dopisz instrukcję, która wydrukuje ci na ekran wartość zmiennej `a` i `b`.

```
void fun1(int a)
{
   a = 2 * a;
}

void fun2(int &a)
{
   a = a + 3;
}

int fun3(int a)
{
   return 3 * a;
}
```

```
void fun4(int *a)
{
   *a = 8;
}

void main()
{
   int a = 5;

   fun1(a);
   // Wydrukuj tu wartosc a
   fun2(a);
   fun3(a);
   a = fun3(a);
   fun4(&a);
   fun2(a);
}
```

Poćwicz jeszcze samodzielnie, aby nabrać przekonania i pewności, że rozumiesz ten mechanizm.

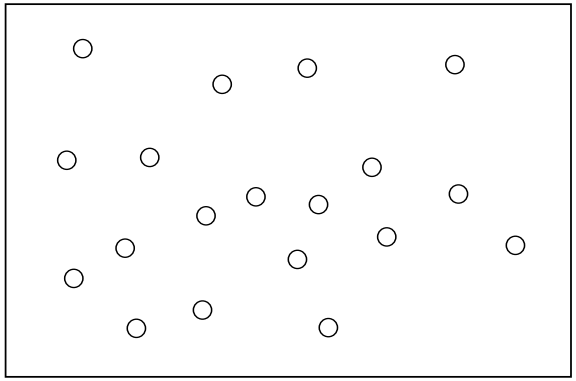
Informatyka I: Instrukcja 5

Tablice

Celem zajęć jest wprowadzenie do używania tablic w języku C. Tablicą (ang. *array*) nazywamy ciąg zmiennych zgromadzony pod jedną globalną nazwą, które są identyfikowane indeksami. Na tych zajęciach zajmiemy się tylko tablicami statycznymi tzn. takimi, których rozmiar jest określany w momencie deklaracji[4]. Tablicę statyczną deklarujemy tak, jak zwykłą zmienną, przy czym dodatkowo określamy jej długość. Wszystko wygląda, jak w przykładowym kodzie poniżej:

```
double a[4];     // deklaracja tablicy

a[0] = 5.5;    // przypisanie wartości do zmiennych
a[1] = 3.521;
```

[4]bardziej zaawansowany mechanizm alokacji tablic będzie tematem następnych zajęć



Rysunek 7: Piłki w ramce.

```
a[2] = 6.45;
a[3] = 4.51;
```

Zwróć uwagę, że elementy tablicy sa indeksowane od 0 do $n-1$, gdzie n to rozmiar tablicy. Można również zainicjalizować wszystkie elementy tablicy natychmiast (taki mechanizm jest użyteczny, jeśli wektory są stosunkowo krótkie):

```
double b[3] = { 1.2, 2.4, -4.3};
```

Piłki

Zadanie polegać będzie na wygenerowaniu zestawu piłek w oknie graficznym oraz na wyszukiwaniu i określaniu ich specyficznych cech, jak np. minimalna, średnia i maksymalna pozycja, maksymanla masa itp. Przykładowy zbiór takich piłek jest widoczny na Rysunku 1.

14 Inicjalizacja

Nasze piłki będą przychowywane tylko jako zestawy współrzędnych, ich prędkości oraz masy. Gdy będziemy chcieli obejrzeć piłki w oknie graficznym, po prostu użyjemy funkcji `circle`. Toteż w symulacji będą potrzebne następujące

wektory[5]:

```
double x[10],y[10];            // współrzędne piłek
double vx[10], vy[10];         // składowe prędkości piłek
double m[10];                 //masy piłek
```

Pętla `for`

Większość operacji na tych zmiennych będziemy wykonywać, używając funkcji, które będą przyjmować wprowadzone wyżej wektory jako argumenty. Funkcje będą musiały mieć podaną długość wektorów tak, aby można było wykonać pewne operacje dla każdego z elementów tego wektora. Jeśli chcemy np. zainicjalizować wszystkie współrzędne wartością 0, piszemy funkcję następującej treści:

```
void init(double *x, double *y, int N){

      for ( int i=0; i < N; ++i){
          x[i] = 0.0;
          y[i] = 0.0;}
}
```

Wykorzystaliśmy tutaj pętlę `for`, która pobiera 3 argumenty:

- wartość startową,
- warunek działania (pętla działa, dopóki warunek $i < N$ jest spełniony),
- operację na argumencie (tutaj zwiększamy i o 1, co będzie najpowszechniejszą praktyką[6])

Taką funkcję wywołujemy w programie głównym, podając nazwy wektorów, na których ma ona działać oraz długość tych wektorów:

```
init(x, y, 10);
```

[5] tablice często będziemy nazywać wektorami, ze względu na fakt, że określenie "tablica" kojarzy się z obiektem o większej ilości wymiarów np. z macierzą

[6] Teoretycznie możemy w tym miejscu wykonać dowolną operację, jednak dla czytelności kodu zazwyczaj zwiększamy licznik pętli

Zauważmy, że funkcja `init` pobiera 2 wskaźniki do wektorów(`x` oraz `y`) oraz jedną wartość (10). Dzięki temu funkcja operuje bezpośrednio na wektorach, na których ma operować i niczego nie musi zwracać[7].

Uwaga

Ponieważ `x` oraz `y` są wskaźnikami do pierwszych elementów tablic, można użyć mechanizmu wyłuskania wartości ze wskaźników i iterować się po wskaźnikach. Poniższy fragment kodu pokazuje dwa równoważne sposoby dostępu do wartości z tablicy:

```
double a[3];
// po wartościach:
a[0] = 1.2;         a[1] = 3.13;         a[2] = 0.22;
//albo na wskaźnikach:
*(a)   = 1.2;           *(a+1) = 3.13;        *(a+2) = 0.22;
```

Ćwiczenia

Przed wykonaniem ćwiczeń upewnij się, że załączono bibliotekę `winbgi2.h`, gdyż będziemy korzystać z grafiki.

1. Zadeklaruj wymienione wyżej wektory położeń, prędkości oraz mas. Utwórz 20 piłek.
2. Zadeklaruj okno graficzne o wymiarach $Lx \times Ly$.
3. Napisz funkcję `init(double *x, double *y, double *vx, double *vy)`, która wylosuje współrzędne położeń początkowych tak, aby powstałe piłki mieściły się w oknie graficznym a składowe prędkosci zawierały się w przedziale $\langle -20, 20 \rangle$. Użyj funkcji `rand()` znanej z poprzednich zajęć.
4. Wypisz na ekran współrzędne wszytkich piłek oraz ich prędkości.
5. Napisz funkcję `display(double *x, double *y, int N)`, która wyświetli położenie i prędkości piłek. Wyświetl piłki.
6. Każdej piłce przypisz masę - niech piłka o indeksie i ma masę $m_i = 2i^2+1$.

[7] Zasady działania na wskaźnikach opisano w Instrukcji 4.2.

15 Analiza położeń i mas

Często nasz zestaw danych musimy poddać jakiejś analizie. Np. chcielibyśmy wiedzieć, która piłka ma największą współrzędną y. W tym celu musimy dokonać przeszukiwania w danym zbiorze danych. Ponadto chcielibyśmy wykonać taką operację możliwie niskim kosztem. W tym przypadku możemy zrobić to, iterując się tylko raz po całym zbiorze piłeczek. Dodatkowo utworzymy tylko tymczasową zmienną (bufor), która będzie przechowywać maksymalna wartość współrzędnej y. Ten bufor nazwiemy `ymax` , gdyż po działaniu pętli to w nim zostanie wartość maksymalna. To wszystko ilustruje poniższy kawałek kodu:

```
double ymax=0.0; //od czegoś trzeba zacząć

for( i=0; i < N; ++i){
     if(y[i] > ymax){
          ymax=y[i];}
}
```

Powyższa pętla kroczy po wszystkich współrzędnych y i jeśli któraś z nich jest większa od wartości aktualnie znajdującej się w buforze, jej y staje się nowym maksimum. Nie trudno zauważyć, że jeden taki cykl załatwia sprawę do końca. Oczywiście w tej samej pętli moglibyśmy zrobić inne interesujące nas rzeczy. Np. możemy zabrać się za liczenie średniej masy piłek:

```
double m_avg = 0.0;;

for( i=0; i < N; ++i){
         m_avg+=m[i]; //liczymy całkowitą masę piłek
}

m_avg = m_ang/N; //liczymy średnią
```

Ćwiczenia

1. Napisz fragment kodu, który znajduje piłki o minimalnej i maksymalnej współrzędnej x oraz y. Na ekranie wypisz te minima i maksima wraz z indeksami tych piłek.
2. Napisz funkcję `crossOut(double *x, double *y, int i)`, która przekreśli krzyżykiem piłkę o indeksie i. Do skreślania użyj dwóch funkcji `line`. Za pomocą tej funkcji skreśl piłki z poprzedniego podpunktu.
3. Napisz fragment kodu, który policzy minimalną, średnią i maksymalną energię kinetyczną piłek. Wypisz te wartosci na ekran. Energię kinetyczną liczymy ze wzoru $E_k = 1/2m(v_x^2 + v_y^2)$.
4. Na środku ekranu narysuj koło o promieniu równym 0.3 przekątnej okna graficznego. Następnie wykreśl wszystkie piłki, których środki nie znajdują się w obszarze tego koła.
5. Napisz funkcje `swapX` i `swapY`, które zamienią wskazane współrzędne piłek w zbiorze wg schematu pierwsza z ostatnią, druga z przedostatnią itd. Zamiana powinna odbyć się w obrębie jednej petli! Wyświetl nowy zbiór piłek po każdej zamianie.

16 *Zaawansowane przeszukiwanie

Do tej pory mieliśmy do czynienia z iteracjami, których liczba rosła liniowo (tzn. proporcjonalnie) do rozmiaru zbioru. Co gdybyśmy chcieli np. porównywać piłki między sobą? Wówczas musielibyśmy iterować się po wszystkich parach piłek. Np. jeśli piłek jest 10, par jest 45. Taki proces ma koszt kwadratowy (dokładnie $n(n-1)/2$). W poniższym przykładzie znaleźć maksymalną odległość między dwoma piłkami. Odległość między dwoma piłkami będzie dana wzorem:

$$L = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

W najprotszej wersji możemy najpierw stworzyć tablicę, która przechowa wszystkie możliwe odległości, a następnie znaleźć jej maksimum. Jednak należy zrobić to bardziej elgancko, bez deklarowania dodatkowych tablic. Załatwi to następujący kod:

```
double L_max=0.0;
double L_tmp;        //dodatkowa zmienna dla czytelności kodu

for(i =0; i < 10; +i){
     for(j=i+1; j < 10; ++j){
          L = sqrt((x[i]-x[j])*(x[i]-x[j])
                  + (y[i]-y[j])* (y[i]-y[j]));
          if(L>L_max){
                   L_max = L;}
          }
}
```

Zauważmy, że podpętla po indeksie j ma zakres zależny od i: nie ma sensu przeszukiwać wszystkich piłek wstecz. Wystarczy, że każda piłka o indeksie i policzy swoją odległość do piłek, których indeksy następują po niej samej.

Ćwiczenia

1. Znajdź najmniejszą i najwiekszą odległość między piłkami. Wypisz te odległości oraz indeksy tych piłek na ekran.
2. Narysuj linie łączące 2 najbliższe piłki i 2 najdalsze.
3. Połącz liniami piłki, które są od siebie dalej niż $Lx/2$.

INFORMATYKA I: INSTRUKCJA 5+

Tablice

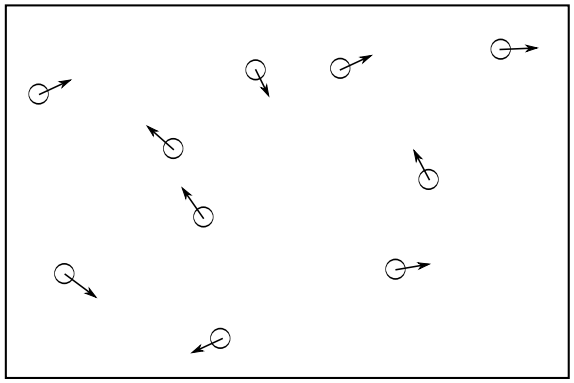
Celem zajęć jest wprowadzenie do używania tablic w języku C. Tablicą (ang. *array*) nazywamy ciąg zmiennych zgromadzony pod jedną globalną nazwą, które są identyfikowane indeksami. Na tych zajęciach zajmiemy się tylko tablicami statycznymi tzn. takimi, których rozmiar jest określany w momencie deklaracji[8]. Tablicę statyczną deklarujemy tak, jak zwykłą zmienną, przy czym dodatkowo określamy jej długość. Wszystko wygląda, jak w przykładowym kodzie poniżej:

```
double a[4];     // deklaracja tablicy

a[0] = 5.5;    // przypisanie wartości do zmiennych
a[1] = 3.521;
a[2] = 6.45;
a[3] = 4.51;
```

Zwróć uwagę, że elementy tablicy sa indeksowane od 0 do $n-1$, gdzie n to rozmiar tablicy. Można również zainicjalizować wszystkie elementy tablicy natychmiast (taki mechanizm jest użyteczny, jeśli wektory są stosunkowo krótkie):

```
double b[3] = { 1.2, 2.4, -4.3};
```

[8] bardziej zaawansowany mechanizm alokacji tablic będzie tematem następnych zajęć



Rysunek 8: Kulki wraz z wektorami predkości początkowych.

Gra w kulki

Zadanie polegać będzie na wygenerowaniu zestawu małych piłeczek w oknie graficznym, wprawieniu ich w ruch oraz implementacji prostych zasad kolizji. Ekran początkowy jest widoczny na Rysunku 1.

17 Inicjalizacja

nasze piłki będą przychowywane tylko jako zestawy współrzędnych oraz ich prędkości. Gdy będziemy chcieli obejrzeć piłki w oknie graficznym, po prostu użyjemy funkcji `circle`. Toteż w symulacji będą potrzebne następujące wektory[9]:

```
double x[10],y[10]           // współrzędne piłek
double cx[10], cy[10]      // składowe prędkości piłek
```

Pętla `for`

Większość operacji na tych zmiennych będziemy wykonywać, używając funkcji, które będą przyjmować wprowadzone wyżej wektory jako argumenty. Funkcje

[9] tablice zazwyczaj będziemy nazywać wektorami, ze względu na fakt, że określenie "tablica"kojarzą się z obiektem o większej ilości wymiarów np. z macierzą

będą musiały mieć podaną długość wektorów tak, aby można było wykonać pewne operacje dla każdego z elementów tego wektora. Jeśli chcemy np. zainicjalizować wszystkie współrzędne wartością 0, piszemy funkcję następującej treści:

```
void init(double *x, double *y, int N){

      for ( int i=0; i < N; ++i){
          x[i] = 0.0;
          y[i] = 0.0;}
}
```

Wykorzystaliśmy tutaj pętlę `for`, która pobiera 3 argumenty:

- wartość startową,
- warunek działania (pętle działa, dopóki warunek $i < N$ jest spełniony),
- operację na argumencie (tutaj zwiększamy i o 1, co będzie najpowszechniejszą praktyką[10]).

Taką funkcję wywołujemy w programie głównym, podając nazwy wektorów, na których ma ona działać oraz długość tych wektorów:

```
init(x, y, 10);
```

Zauważmy, że funkcja `init` pobiera 2 wskaźniki do wektorów(`x` oraz `y`) oraz jedną wartość (10). Dzięki temu funkcja operuje bezpośrednio na wektorach, na których ma operować i niczego nie musi zwracać[11].

Uwaga

Ponieważ `x` oraz `y` są wskaźnikami do pierwszych (dokładnie rzecz ujmując - zerowych) elementów tablic, można użyć mechanizmu wyłuskania wartości ze wskaźników i iterować się po wskaźnikach. Poniższy fragment kodu pokazuje dwa równoważne sposoby dostępu do wartości z tablicy:

```
double a[3];
// po wartościach:
a[0] = 1.2;         a[1] = 3.13;         a[2] = 0.22;
//albo na wskaźnikach:
*(a)   = 1.2;           *(a+1) = 3.13;        *(a+2) = 0.22;
```

Ćwiczenia

Przed wykonaniem ćwiczeń upewnij się, że załączono bibliotekę `winbgi2.h`, gdyż będziemy korzystać z grafiki.

1. Zadeklaruj wymienione wyżej wektory o długości 10.
2. Zadeklaruj okno graficzne o wymiarach $Lx \times Ly$.
3. Napisz funkcję `init`, która wylosuje współrzędne położeń początkowych tak, aby powstałe kółka mieściły się w oknie graficznym. Użyj funkcji `rand()` znanej z poprzednich zajęć.
4. Napisz funkcję `display`, która wyświetli położenie kółek (funkcja powina mieć tę samą strukturę, co funkcja `init`).

18 Ruch

Oczywiście piłeczki mają się poruszać, zatem konieczne będzie określenie wartości prędkości początkowych oraz zaprogramowanie ruchu piłeczek.

Ćwiczenia

1. Napisz funkcję, która wylosuje początkowe prędkości piłek. Wylosuj je tak, aby wartość prędkości wynosiła 1 (najłatwiej będzie wylosować dowolną liczbę i jej sinus i cosinus przypisać jako składowe prędkości piłki)
2. Napisz funkcję `run`, która będzie wykonwać przesunięcie każdej z piłek. Przemieszczenie będzie po prostu polegać na zwiększeniu każdej współrzędnej o składową prędkość[12]:

[10] Teoretycznie możemy w tym miejscu wykonać dowolną operację, jednak dla czytelności kodu zazwyczaj zwiększamy licznik pętli

[11] Zasady działania na wskaźnikach opisano w Instrukcji 4.2.

[12] piłki poruszają się ze stałą prędkością, toteż $x(t + \Delta t) = x(t) + v\Delta t$, a dla uproszczenia symulacji czas jest jednostkowy zatem $x(t + 1) = x(t) + v$

```
for ( i=0; i < N; ++i){
     x[i] += cx[i];
     y[i] += cy[i];
}
```

3. W głównym programie napisz pętlę `while`, która wykona 50 kroków iteracji programu. Niech przy każdym kroku wyświetla położenie każdej piłki. W ciele pętli użyj funkcji `animate(100)` - spowolni ona wykonywanie kolejnych kroków pętli. Jej użycie wyglądało następująco.

```
while(animate(100)) {
   clear(); // wyczysci okno graficzne dla nowej klatki
   // Dalsza czesc ciala petli
}
```

19 Kolizje ze ścianami

Chcielibyśmy, aby piłeczki miały wbudowany jakiś mechanizm kolizji ze ścianami. Zderzenia będą doskonale sprężyste, kąt padania na przeszkodę będzie zatem równy kątowi odbicia od niej. Kąty mierzone względem normalnej do ściany.

Ćwiczenia

1. Do funkcji `run` dopisz warunek, który sprawdza, czy piłka zderzyła się ze ścianą. W przypadku kolizji należy zastosować prawo odbicia, które będzie miało prostą formę: Jeśli uderzamy w ścianę poziomą, wystarczy zmienić składową prędkości cy na przeciwną. Analogicznie przy kolizji ze ścianą pionową, zmieniamy składową cx na przeciwną. Sprawdź, jak działa program np. dla 5000 kroków.

2. Napisz funkcję `showEnergy`, która będzie wyświetlała na ekranie wartość całkowitej energii kinetycznej układu.

20 Kolizje z piłkami*

Dopisz funkcję `searchAndCollide`, która sprawdza, czy piłki zderzają się ze sobą nawzajem. Trzeba będzie przeiterować się po wszystkich współrzędnych sąsiadów i sprawdzić, czy odległość piłek jest dostatecznie mała. Jeśli tak jest, piłki odbiją się od siebie, zachowując pęd oraz energię. Załóżmy, że piłki, które się ze sobą zderzą mają indeksy i i j. Ich prędkości należy policzyć w następujący sposób:

1. Należy sprowadzić wszystko do układu odniesienia zwiazanego z drugą piłeczką oraz dodatkowo policzyć wektor jednostkowy wskazujący kierunek łączący środki obu piłek:

$$v_1 = [cx_i - cx_j, cy_i - cy_j]$$

$$L = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

$$n = [(x_i - x_j)/L, (y_i - y_j)/L]$$

2. Przy zderzeniu przekazywana jest jedynie predkość normalna do płaszczyzny zderzenia obu piłek (patrz Rys. 2 i 3). Liczymy ją w nastepujący sposób:

$$v_n = [v_{nx}, v_{ny}] = [(v_{1x}n_x + v_{1y}n_y)n_x, (v_{1x}n_x + v_{1y}n_y)n_y]$$

3. Policzona powyżej predkość jest odejmowana od prędkości piłki 1 oraz dodawana do prędkosci piłki 2 (która była zerem w nowym układzie):

$$v_1^{new} = [v_{1x} - v_{nx}, v_{1y} - v_{ny}], v_2^{new} = [v_{nx}, v_{ny}]$$

4. Na koniec wracamy do starego układu odniesienia:

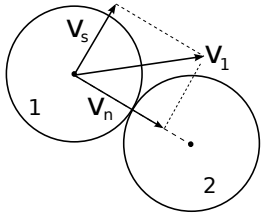
$$cx_i^{new} = cx_i + v_{1x}^{new}$$

$$cy_i^{new} = cy_i + v_{1y}^{new}$$

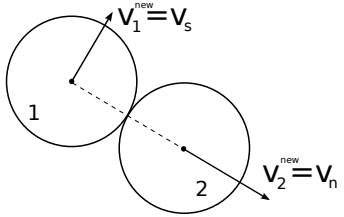
$$cx_j^{new} = cx_j + v_{2x}^{new}$$

$$cy_j^{new} = cy_j + v_{2y}^{new}$$

INFORMATYKA I: INSTRUKCJA 6



Rysunek 9: Zderzenie piłek w układzie związanym z drugą piłką (druga piłka jest nieruchoma). Prędkości przed wymianą pędu..



Rysunek 10: Zderzenie. Prędkosci po wymianie pędu. Składowa równoległa do osi wyzanczonej przez środki piłek zostaje przekazana piłce 2.

21 Współpraca z plikami

Praktyczny program (szczególnie inżynierski) bardzo często musi współpracować z plikami. Czasem, przy obliczeniach trwających wiele godzin lub dni wręcz zależy nam na tym, by program działał samodzielnie bez potrzeby interakcji ze strony użytkownika. Wróćmy jednak do plików. Najczęściej chodzi o możliwość wczytania danych wejściowych z jednego (bądź wielu) plików, przeprowadzenie obliczeń wewnątrz programu i zapisanie wyników do innego pliku (bądź wielu plików).
W języku C komunikacja z plikami prowadzona jest niemalże identycznie, jak czytanie danych z klawiatury i wydruk na ekran, co realizowaliśmy za pomocą znanych już funkcji `scanf` oraz `printf`. Musimy jednak najpierw określić, jaki plik chcemy utworzyć bądź otworzyć i w jakim celu go tworzymy/otwieramy. Ponadto, analogiem instrukcji `printf` do zapisu do plików jest instrukcja `fprintf`, zaś funkcja `scanf` jest zastąpiona przez funkcję `fscanf`. Przyjrzyjmy się przykładowemu kodowi źródłowemu.

```
FILE *f; // deklarujemy zmienną typu FILE o nazwie f
         // tak naprawde to wskaznik (o tym jednak w pozniej)

f = fopen("plik.txt", "w"); // otwieramy plik o nazwie plik.txt
                            // z zamiarem zapisu ("w") i przypi-
                            // sujemy go do zmiennej o nazwie f

fprintf(f, "Zapisujemy wlasnie ten tekst do pliku\n");

fclose(f); // zamykamy plik
```

Warto zwrócić uwagę na trzy kwestie. Po pierwsze, w funkcji `fprintf` (to samo dotyczy funkcji `fscanf` jako pierwszy argument trzeba podać *strumień* - de facto nazwę zmiennej typu `FILE`, z którym zachodzi komunikacja (zapis lub odczyt). Dlatego tu podajemy *f*, bo tak właśnie nazwaliśmy naszą zmienną. Druga uwaga: Powyższy zapis można nieco skompresować (połączyć deklarację zmiennej typu FILE z otwarciem pliku). Ponadto, istotne jest, by sprawdzić, czy plik udało się otworzyć. W przeciwnym razie jakiekolwiek operacje nie miałyby sensu lub skończyły się błędem naszego programu. Zmodyfikujmy więc nasze instrukcje. Teraz wyglądają tak:

```
FILE *f = fopen("plik.txt", "w")

if(f == NULL)
```

```
{
   printf("Blad otwarcia pliku\n");
   exit(-1); // zakonczenie programu
}

// Tu wykonujemy operacje na pliku (w naszym przypadku zapis)
// Gdy plik juz nie bedzie wiecej potrzebny w naszym programie,
// koniecznie go zamykamy!

fclose(f);
```

Pliki można otworzyć nie tylko w trybie zapisu *(ang. write)* `w` (który zawsze czyści plik i wypełnia go od nowa), ale również w trybie dopisywania do pliku *(ang. append)* `a` lub czytania z pliku *(ang. read)* `r`. Można również wybrać, czy tworzony/czytany plik ma być obsługiwany w trybie tekstowym czy binarnym. Służą do tego odpowiednio sekwencje `t` i `b`. Przykładowe instrukcje zaprezentowano poniżej. Zauważmy też, że można otworzyć w tym samym czasie kilka plików.

```
void main()
{
   int a = 3;

   FILE *f = fopen("plik1.txt", "wt"); // Zapis w trybie tekstowym
   FILE *g = fopen("plik2.dat", "wb"); // Zapis w trybie binarnym
   FILE *InnyPlik = fopen("Dane.txt", "r"); // Czytanie z pliku

   if((f == NULL) || (g == NULL) || (InnyPlik == NULL))
   {
      printf("Nie udalo sie otwarcie choc jednego z plikow\n");
      exit(-1);
   }

   fprintf(f, "Zapisujemy wartosc a do plik1.txt, a = %d\n", a);
   fprintf(g, "Binarnie zapisujemy ten tekst do plik2.dat\n");
   fscanf(InnyPlik, "%d", &a); // Wczytujemy z pliku Dane.txt
// liczbe calkowita i przypisujemy jej wartosc do zmiennej a

   // Tu mozemy wykonac jeszcze inne operacje na otwartych plikach

   fclose(f);
   fclose(g);
   fclose(InnyPlik);
}
```

W powyższym przykładzie zaprezentowaliśmy jednocześnie użycie funkcji `fscanf`, która działa analogicznie do dobrze już znanej funkcji `scanf`.

Uwaga

Wszystkie funkcje związane z obsługą plików znajdują się w bibliotece `stdlib.h`. W związku z tym do pliku programu należy dołączyć instrukcję preprocesora załączającą tę bibliotekę: `#include <stdlib.h>`

Ćwiczenia

W praktyce inżynierskiej pliki często zawierają dane pochodzące z eksperymentu lub symulacji. Plik `przebieg.txt` zawiera fragment przebiegu czasowego wartości trzech składowych prędkości *(u, v, w)* pochodzących z symulacji przepływu powietrza przez dużą turbinę wiatrową. Chwilowe wartości tych składowych zostały zebrane z punktu znajdującego się tuż za turbiną. Napisz program, który:

- Otworzy plik.
- Wczyta dane z pliku do trzech tablic `u`, `v`, `w` zadeklarowanych statycznie (każda o rozmiarze 2000 - za tydzień będzie o lepszej metodzie deklaracji dużych tablic). Czytanie zrealizuj z użyciem pętli `for`. Obejrzyj plik, aby przyjrzeć się, w jaki sposób ułożone są dane (każda z kolumn odpowiada jednej ze składowych prędkości *(u, v, w)*; kolejne wiersze odpowiadają kolejnym krokom czasowym).
- Po wczytaniu wszystkich wartości do tablic obliczy średnią każdej ze składowych. Średnia wyrażona jest wzorem:

$$\bar{u} = \frac{\sum_{i=1}^{n} u_i}{n} \tag{1}$$

- Obliczy odchylenie standardowe dla każdej ze składowych. Odchylenie standardowe dane jest wzorem:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(u_i - \bar{u})^2}{n-1}} \tag{2}$$

- Zapisze do innego pliku raport z obliczeń, w którym poda wszystkie obliczone wielkości oraz wydrukuje to samo na ekran.
- Wczytaj też plik `przebieg.txt` do arkusza kalkulacyjnego i utwórz wykres obrazujący te przebiegi. Oceń krytycznie wyniki uzyskane swoim programem na podstawie obserwacji wykresu. Czy średnie i odchylenia standardowe mają wiarygodne wartości?

Wskazówka

Zauważ, że każda suma daje się łatwo policzyć z użyciem pętli `for` w następujący sposób:

```
double suma = 0;

for(int i = 0; i<n; i++)
{
   suma += a[i];
}
```

To tylko wskazówka. Oczywiście musisz zmodyfikować powyższy kod tak, aby liczył sumy z powyższych wzorów.

22 Ważne: Dalej o funkcjach

Wiemy, że funkcje mogą przyjmować argumenty. Dowiedzieliśmy się też, że funkcje mogą zwracać wartości. Zmodyfikuj swój kod tak, aby odpowiednie bloki instrukcji były realizowane w funkcjach `Srednia` i `OdchylenieStandardowe`. Powinny mieć takie nagłówki:

```
double Srednia(double *tablica, int n);
double OdchStd(double *tablica, double WartoscSrednia, int n);
```

Następnie zmodyfikuj kod funkcji `main` tak, aby część dotycząca obliczeń dała się zwięźle zapisać w poniższej postaci:

```
void main()
{
   (...) // Deklaracje i kod wczytujacy dane

   um = Srednia(u, n);
   vm = Srednia(v, n);
   wm = Srednia(w, n);

   u_std = OdchStd(u, um, n);
   v_std = OdchStd(v, vm, n);
   w_std = OdchStd(w, wm, n);

   (...) // Dalsza czesc programu zajmujaca sie raportowaniem wynikow
}
```

*Dla dociekliwych

Obliczenia na komputerze prowadzone są ze skończoną dokładnością. Zmodyfikuj swój kod tak, aby bieżąca wartość średniej była liczona „w locie" - w trakcie czytania danych z pliku (naturalnie będzie to średnia wartość przeczytanych dotąd elementów). Wystarczy, że zrobisz to dla jednej składowej prędkości (np. u). Możesz tę średnią też na bieżąco podczas czytania danych drukować na ekran. Na końcu porównaj wartość średniej uzyskanej w ten sposób z wartością policzoną *a posteriori* w poprzednim poleceniu.
Pseudokod algorytmu znajdziesz poniżej. Zapisz go w sposób zrozumiały dla komputera, w języku C.

```
biezaca_srednia = 0

Petla po i od 1 do n (czytajaca dane)
{
   PrzeczytajNowyElementZPlikuIWpiszGoDoTablicy

   biezaca_srednia = (biezaca_srednia*(i-1) + u[i])/i
}

// Po zakonczeniu petli biezaca_srednia to srednia z calego zbioru
```

Zastanów się, dlaczego taki algorytm liczenia średniej w sensie matematycznym prowadzi do tak samo zdefiniowanej średniej. Jeśli trudno ci go zrozumieć, wymyśl sobie zbiór czteroelementowy i wykonaj go krok po kroku na kartce.

Pytanie

Czy obie średnie (policzone na komputerze dwoma sposobami) mają tę samą wartość? Czy coś się zmieni, gdy weźmiesz inną składową prędkości?

Informatyka I: Instrukcja 7

Dynamiczna alokacja pamięci

Do tej pory używaliśmy tablic, które miały z góry zadany rozmiar. Deklarowaliśmy je w bardzo prosty sposób:

```
double x[100];
```

Oczywiście chcielibyśmy poszerzyć funkcjonalność naszego programu tak, aby była możliwa deklaracja tablic o długości określonej już w trakcie działania programu. Mówimy wówczas o tzw. dynamicznej (tzn. wykonywanej podczas działania programu, a nie podczas kompilacji) alokacji pamięci. Taki mechanizm jest jak najbardziej dopuszczalny w języku C, składa się na niego kilka etapów:

1. Deklaracja wskaźnika do nowotworzonej tablicy.
2. Alokacja pamięci dla danej tablicy - używa się w tym celu funkcji `malloc()`.
3. Zwolnienie pamięci - funkcja `free()`.

Nowe funkcje wymagają użycia biblioteki `stdlib.h`. Poniższy kawałek kodu ilustruje cały mechanizm:

```
#include <stdlib.h>

void main()
{
     int N=100;
     double *x;        // wskaźnik do pierwszego elementu tablicy

     x = (double*)malloc(N*sizeof(double));       // alokacja

     free(x);          // zwolnienie pamięci
}
```

Kluczowym elementem całego mechanizmu jest właściwe zastosowanie funkcji `malloc`. Jej argumentem jest rozmiar pamięci, o którą wnioskujemy. Potrzebujemy N razy rozmiar zajmowany przez zmienną typu `double` - stąd obecność funkcji `sizeof()`, która zwraca rozmiar danego typu zmiennych. Ponadto używamy mechanizmu rzutowania - przed funkcją `malloc` pojawia się rzutowanie na typ `double` poprzez wywołanie `(double*)`[13]. Po pracy na tablicy należy pamiętać o zwolnieniu pamięci, którą ona zajmowała. Stąd funkcja `free()`.

Wczytywanie i rysowanie konturów

Nowy mechanizm będzie nam potrzebny do wykonania zadania polegającego na wczytaniu współrzędnych konturów map z plików oraz narysowaniu tych map na ekranie. Współrzędne konturów dwóch map są zapisane w dwóch plikach, `plik1.txt` oraz `plik2.txt`. Pliki mają następującą strukturę:

```
156
12.67  768.3254
14.98  768.3254
17.462 766.51075
...
```

Pierwsza linia pliku zawiera ilość punktów zapisanych w pliku. Poniżej współrzędne x oraz y są zapisane w oddzielnych kolumnach.

Ćwiczenia

1. Otwórz oba pliki w programie głównym.

[13] programując w czystym C można by teoretycznie zrezygnować z rzutowania, wówczas `malloc` zwróci wskaźnik typu `void*`. Jednak powszechnie stosujemy kompilatory języka C++, który jest językiem o tzw. silnej kontroli typów i nie zezwala na to, by wskaźnik nie miał typu. Nie zagłębiając się bardziej w szczegóły, należy po prostu wyrobić w sobie nawyk rzutowania przed użyciem funkcji `malloc()`, aby uniknąć wielu nieprzyjemnych sytuacji w przyszłości.

2. Wczytaj pierwsze linie obu plików do programu. Wydrukuj na ekran te wartości, aby upewnić się, że wykonano ten podpunkt bezbłędnie.

3. Użyj wczytanych wartości do alokacji tablic, które będą przechowywać współrzędne punktów. Użyj mechanizmu alokacji dynamicznej.

4. Wczytaj współrzędne z obu plików do wcześniej zadeklarowanych tablic. Upewnij się, czy wykonano to dobrze.

5. Napisz funkcję `void PrintCoords(double *x, double *y, int N,int start, int end)`, która będzie drukować na ekranie zadany przedział współrzędnych z danej mapy. Funkcja ma informować, gdy zadany przedział nie może być wydrukowany (bo np. kres górny przekracza długość tablicy)

6. napisz funkcję `void Display()`, która będzie rysować zadany kontur mapy na ekranie w oknie graficznym. Kontur ma być narysowany za pomocą linii, które będą łączyć kolejne punkty. Zwroć uwagę na to, by połączyć również ostatni punkt z pierwszym. Wyświetl oba kontury na jednym obrazku. Co to za mapy?

Trochę geografii

Mając już kontur, możemy policzyć kilka interesujących rzeczy:

Ćwiczenia

1. Napisz funkcję `double Perimeter()`, która zwróci obwód zadanego konturu.

2. Napisz funkcję `double Area()`, która policzy oraz zwróci powierzchnię danej mapy. Powierzchnię mapy należy policzyć jako sumę powierzchni trójkątów, których podstawy są kolejnymi odcinkmi konturu a wierzchołek jest zlokalizowany gdziekolwiek (najwygodzniej położyć go w zerze). Pole takiego trójkąta będzie po prostu połową modułu iloczynu wektorowego, gdzie wektorami są boki trójkąta:

$$S_i = \frac{1}{2}|\boldsymbol{v}_i \times \boldsymbol{v}_{i+1}| = \frac{1}{2}(x_i y_{i+1} - x_{i+1} y_i)$$

   Pamiętaj o ostatnim i pierwszym punkcie. Zwróć uwagę, że otrzymane pole może być ujemne. Od czego to zależy?

3. Policz rozciągłość południkową i równoleżnikową obu obszarów. Zaznacz najbardziej skrajne punkty na obu mapach za pomocą kółka.

Informatyka I: Instrukcja 8

Dynamiczna alokacja tablic wielowymiarowych

Tydzień temu nauczyliśmy się dynamicznej alokacji pamięci dla tablic jednowymiarowych. Dynamiczna oznacza tyle, że rozmiar tablicy, jaką chcemy zaalokować, znamy dopiero w momencie wykonywania programu, tzn., że nie jesteśmy w stanie go określić statycznie (daną, konkretną liczbą) na etapie kompilacji. Kod do dynamicznej alokacji tablic jednowymiarowych wyglądał tak:

```
#include <stdlib.h>

void main()
{
   double *tab;
   int n;
   printf("Podaj n:\n");
   scanf("%d, &n);

   tab = (double*)malloc(n*sizeof(double));
   // Tu mozesz wykonywac operacje na tablicy
   free(tab);
}
```

Dziś nauczymy się dwóch nowych rzeczy: stosowania w języku C tablic wielowymiarowych (alokowanych statycznie) oraz ich dynamicznej alokacji.

Tablice wielowymiarowe

Język C pozwala na stosowanie tablic wielowymiarowych. Do tej pory przez kilka tygodni używaliśmy jedynie tablic jednowymiarowych. Potocznie często określaliśmy je *wektorami*. Wyobraźmy sobie - tablica dwuwymiarowa doskonale nadaje się np. do przechowywania macierzy.[14] Przyjrzyjmy się więc frag-

[14] Tak naprawdę wiele więcej struktur - często nawet zupełnie niematematycznych możemy trzymać w dwuwymiarowych tablicach - np. programując grę w szachy moglibyśmy użyć

mentowi kodu, który zadeklaruje dwuwymiarową tablicę o wymiarze 3x4. Możemy to utożsamić z reprezentacją macierzy o takim samym wymiarze.

```
void main()
{
   double A[3][4];

   // Tu mozemy przypisac wartosci kolejnym elementom:
   A[0][0] = 1;
   A[0][1] = 1.5;
   A[0][2] = 0;
   A[0][3] = -2.7;
   ...
   A[2][3] = 8;
   // Tu mamy wypelniona macierz
   // Mozemy wykonywac obliczenia
}
```

Zauważmy, że w przypadku tablic deklarowanych statycznie nie ma potrzeby ich zwalniania. Kompilator sam o to dba (jak w przypadku wszystkich zmiennych, które do tej pory deklarowaliśmy - one też są automatycznie niszczone przez kompilator). Powyższą macierz możemy też wypełnić wartościami w nieco zgrabniejszy sposób niż przez wypisanie kolejnych dwunastu linijek przypisań. Możemy to zrobić na liście inicjalizacyjnej od razu na etapie deklaracji tablicy. Dokonuje się tego tak:

```
double A[3][4] = {{1, 1.5, 0, -2.7}, {-3, 2.5, 7, 0}, {0, 1, -3, 8}};
```

W ten sposób stworzymy poniższą macierz:

$$\mathbf{A} = \begin{pmatrix} 1 & 1.5 & 0 & -2.7 \\ -3 & 2.5 & 7 & 0 \\ 0 & 1 & -3 & 8 \end{pmatrix} \tag{3}$$

dwuwymiarowej tablicy o wymiarze 8 x 8 i w odpowiednie pola tej tablicy wpisywać liczby, które symbolizowałyby konkretne figurę. A gra w statki? Można podobnie. Z drugiej strony w praktycznych obliczeniach numerycznych wielkie macierze o rozmiarze rzędu kilkuset tysięcy do kilku milionów i większe przechowuje się w postaci wektora. W praktycznych zagadnieniach są to niemal zawsze macierze rzadkie, tzn. takie, które w stosunku do całkowitej liczby swoich elementów mają bardzo niewiele elementów, które nie są zerem. Taką macierz łatwo jest trzymać w pamięci jako wektor (przyjmując specjalny format, który pomija wszystkie zera - np. tzw. format CSR *(ang. compressed sparse row)* jest szeroko stosowanym formatem zapisania macierzy rzadkiej w trzech wektorach). Tak wielka macierz przechowywana jawnie najpewniej nie zmieściłaby się w pamięci żadnego dostępnego nam komputera.

Pozostaje wytłumaczyć jeszcze, w jaki sposób odwołujemy się do elementów w dwuwymiarowej tablicy. Robimy to analogicznie do tablicy jednowymiarowej, tylko tym razem musimy podać dwa indeksy. Tak więc do elementu macierzy a_{32} odwołamy się przez napisanie `a[2][1]`. W ten sposób możemy wyłuskać wartość przechowywaną pod tym elementem lub operatorem = przypisać temu elementowi nową wartość.

Ćwiczenia

W funkcji `main` napisz fragment kodu, w którym zadeklarujesz i zainicjalizujesz dowolnymi wartościami dwie różne tablice dwuwymiarowe. Jedna ma przechowywać macierz kwadratową o wymiarze 2, a druga z nich macierz kwadratową o wymiarze 3. Napisz kod, który dla każdej z tych macierzy policzy wyznacznik.

Dynamiczna alokacja

Czas jednak na dynamiczną alokację. Dwuwymiarową tablicę o rozmiarze M x N zaalokujemy w następujący sposób: stworzymy tablicę jednowymiarową o rozmiarze M (umówmy się, że ona będzie wskazywać na początek każdego z wierszy), po czym każdemu z elementów tej tablicy zaalokujemy blok o długości N (to będą jednowymiarowe tablice do przechowywania kolejnych wierszy). De facto będziemy mieli w pamięci M bloków, każdy długości N. Spójrzmy na kod.

```
double **A;
*A = (double**)malloc(M*sizeof(double*));
```

Zauważmy, że tydzień temu alokowaliśmy jednowymiarową tablicę jako wskaźnik. Tym razem będziemy mieć tablicę dwuwymiarową, więc używamy podwójnego wskaźnika. Dlatego w instrukcji powyżej blok pamięci zwracany przez funkcję `malloc` rzutujemy na podwójny wskaźnik `double**`. Musimy też obliczyć, ile miejsca potrzebujemy. Przechowywać będziemy wskaźniki (do odpowiednich tablic jednowymiarowych przechowujących wiersze), dlatego jako argument funkcji `sizeof` podajemy `double*`. Teraz alokujemy tablice jednowymiarowe do przechowywania wierszy.

```
for(int i = 0; i<M; ++i)
   A[i] = (double*)malloc(N*sizeof(double));
```

Powyżej każdemu z elementów pierwszej tablicy przypisaliśmy tablicę do przechowywania każdego z wierszy. Tym razem blok zwrócony przez funkcję `malloc`

rzutujemy na typ `double*`, a argumentem funkcji `sizeof` jest typ zmiennej przechowywanej w tej tablicy, czyli już zwykła zmienna `double`, a nie wskaźnik do niej. Tablica dwuwymiarowa jest już gotowa - możemy jej używać.
Po zakończeniu pracy z tablicą, trzeba koniecznie zwolnić pamięć przez nią wykorzystywaną. Teraz trzeba operacje wykonać od końca! Tzn. najpierw zwalniamy każdy z wierszy, a na końcu zwolnimy pierwotną tablicę wskaźników do wierszy. Dokonuje tego poniższy kod.

```
for(int i = 0; i<M; ++i)
   free(A[i]);

free(A);
```

Podsumowanie

Zbierzmy wszystkie instrukcje w jednym miejscu. Chcemy zaalokować dwuwymiarową tablicę zmiennych typu `int`. Dokonujemy tego tak:

```
// Alokacja pamieci
int **A;
*A = (int**)malloc(M*sizeof(int*);
for(int i = 0; i<M; ++i)
   A[i] = (int*)malloc(N*sizeof(int));

// Tu mozemy wykonywac dowolne operacje na tablicy

// Zwolnienie pamieci
for(int i = 0; i<M; ++i)
   free(A[i]);
free(A);
```

Ćwiczenia

1. Napisz funkcję, która jako argumenty przyjmie podwójny wskaźnik (wskaźnik do dynamicznie alokowanej tablicy dwuwymiarowej) oraz liczbę kolumn i wierszy macierzy, po czym dokona wydruku macierzy na ekran w naturalnej postaci, do jakiej jesteśmy przyzwyczajeni dla macierzy.

2. Napisz funkcję `maxAbsAij`, która dla danej macierzy zwróci do funkcji `main` największy co do modułu element tej macierzy oraz jego indeksy i, j.

3. Napisz funkcję `minAbsAij`, która dla danej macierzy zwróci do funkcji `main` najmniejszy co do modułu element tej macierzy oraz jego indeksy i, j.

4. W funkcji main zaalokuj w sposób dynamiczny miejsce dla macierzy 3x3 oraz dwóch wektorów trzyelementowych. Wypełnij macierz i jeden z wektorów dowolnymi wartościami, po czym napisz w funkcji `main` kod, który dokona przemnożenia danej macierzy przez dany wektor i wynik mnożenia wpisze do drugiego z wektorów. Mnożenie macierzy przez wektor określone jest wzorem
$$w_i = \sum_{j=0}^{n-1} a_{ij} v_j \tag{4}$$
.

5. Zamknij powyższe operacje w funkcji o nagłówku

   ```
   void MatVecMultiply(double **A, double *v1, double *v2, int n)
   ```

   i dokonaj wywołania z funkcji `main`.

6. Zmodyfikuj powyższy program tak, żeby rozmiar n był wczytywany z klawiatury, elementy tablicy były generowane zgodnie ze wzorem $a_{ij} = \frac{i+1}{j+1}, (i,j = 0,...,n-1)$, zaś elementy wektora wg wzoru $v_i = i+1, (i = 0,...,n-1)$. Obliczaj iloczyn takiej macierzy przez ten wektor, korzystając ze swojej funkcji `MatVecMultiply`. Wynik wyświetlaj na ekranie oraz sprawdź, czy otrzymujesz poprawny wynik.[15]

7. Napisz funkcję

   ```
   MatMatMultiply(double **A, double **B, double **C, int mA,
                                          int nA, int mB, int nB)
   ```

[15] Dla takiej macierzy i takiego wektora bardzo łatwo jest wygenerować analityczny wynik. Wypisz sobie małą macierz wg zadanego wzoru i odpowiadający wektor i na pewno szybko zauważysz prawidłowość. Będziesz wiedzieć, jaki wynik powinien dać program. Tak się testuje programy na wczesnych etapach rozwoju.

służącą do mnożenia dwóch macierzy prostokątnych (**A** o wymarze $m_A \times n_A$ i **B** o wymiarze $m_B \times n_B$) i wpisującą wynik do macierzy **C** (zadbaj w funkcji `main` o to, aby pamięć zaalokowana dla macierzy **C** była odpowiedniej wielkości - zgodnej z regułami mnożenia macierzy). **Uwaga:** Pamiętaj, aby koniecznie zwolnić wszelką dynamicznie alokowaną pamięć. [16]

Dla ambitnych*

Zastanów się, jak wyglądałaby dynamiczna alokacja i zwolnienie pamięci dla tablicy trójwymiarowej. Na przykład, gdybyś chciał napisać grę w trójwymiarowe kółko i krzyżyk. Skonsultuj z prowadzącym kod dla takiego przypadku.

INFORMATYKA I: INSTRUKCJA 9

23 Oblężenie

Do poprzednich zajęć poznaliśmy już wszystkie instrukcje zaplanowane na ten kurs języka C. Czas przejść do kilku nieco bardziej dojrzałych przykładów. Dziś zajmiemy się oblężeniem zamku. Twoim zadaniem jest napisać program, który będzie symulował ostrzał zamku przez kule armatnie (kule lecą wg wzorów na rzut ukośny). Program powinien kolejno:

- Przygotować okno graficzne o wymiarach 800 x 600.
- Narysować zamek, który będziemy burzyć (kod funkcji rysującej zamek możesz znaleźć na końcu instrukcji).
- W pętli:
  - wczytywać z klawiatury wartość prędkości wylotowej kuli V_0, kąta odchylenia lufy od poziomu α oraz położenie armaty wzdłuż współrzędnej x (wylot armaty zawsze ma się znajdować 50 jednostek powyżej poziomu ziemi).
  - ustawiać i rysować schematycznie armatę jako jedną linię (należy przy tym sprawdzać, czy w danym miejscu da się postawić armatę - mur zamku znajduje się w obszarze $x \geq 550$; dla uproszczenia radzimy przy tym rysować armatę, jako linię o górnym wierzchołku w punkcie *(x, 50)* i dolnym wierzchołku zależnym od kąta wychylenia).
  - wykonywać strzał kulą i rysować jej tor za pomocą gęsto rozłożonych okręgów o małym promieniu - użyj funkcji `animate` w celu uzyskania efektu płynnego ruchu kuli (przykład użycia funkcji `animate` na końcu instrukcji). Tor rzutu ukośnego dany jest wzorami:

$$x(t) = x_0 + V_0 sin(\alpha)t \tag{5}$$

$$y(t) = y_0 + V_0 cos(\alpha)t - \frac{gt^2}{2}. \tag{6}$$

    Program powinien cały czas w pętli sprawdzać, czy kula nie uderzyła w pierwszą, pionową ścianę zamku. Jeśli tak, kula ma się na niej zatrzymać, a pętla ulec przerwaniu. W przeciwnym razie program ma tak długo rysować ruch kuli, aż ta znajdzie się na ujemnych wartościach współrzędnej y. Ściana zamku zawiera się w przedziale $550 \leq x \leq 600$ oraz jej górna powierzchnia ma współrzędną $y = 300$. Pamiętaj również o przeliczeniu współrzędnych fizycznych (pionowej) na współrzędną w układzie współrzędnych ekranu oraz przeliczeniu kąta podawanego w stopniach na kąt podany w radianach na potrzeby funkcji trygonometrycznych.
  - Po wykonanym strzale program zapyta, czy udało się trafić w zamek. Jeśli odpowiesz twierdząco, zakończy działanie programu. Jeśli przecząco, pozwoli wybrać nowe parametry strzału i będzie powtarzał tę operację aż do chwili uzyskania satysfakcjonującego cię wyniku.

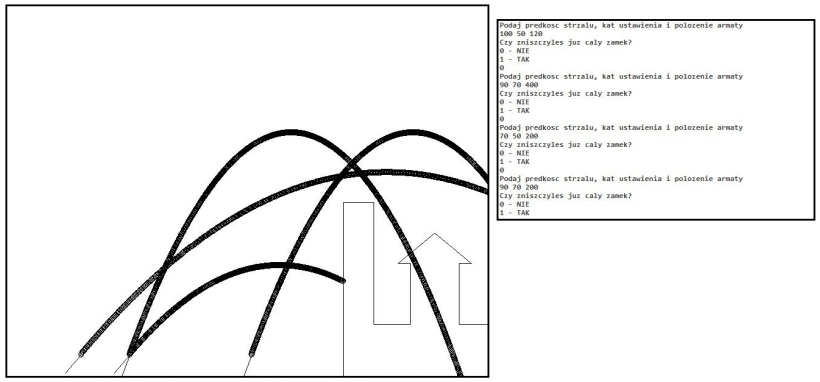
Wskazówka

W niniejszym programie w ogóle nie używaj funkcji `wait()`. Program z wykonaniem i tak będzie czekać na podanie nowego zestawu danych do funkcji `scanf()`.

Oczekiwany wynik

Spodziewamy się, że wynikiem działania twojego programu będzie efekt zbliżony do tego pokazanego na Rysunku 1.

[16] Niezwolnienie pamięci prowadzi do jej wycieków i w przypadku pewnych operacji wykonywanych w pętlach może doprowadzić do tego, że Twój program wykorzysta całą pamięć operacyjną komputera i przestanie działać.



Rysunek 11: Oczekiwany efekt działania programu

Dodatki

Funkcja `animate` tak naprawdę jedynie spowalnia wykonywanie pętli `while` tak, aby kółka nie rysowały się zbyt szybko. Przykład jej użycia wygląda następująco:

```
while(animate(100)) // Wartosc 0 oznacza najszybsze mozliwe
                    // wykonanie (brak opoznienia)
{
   // Oblicz tu nowe wspolrzedne x i y
   // Narysuj okrag w odpowiednim miejscu
   // Zwieksz parametr t
   // Zawrzyj warunek przerwania petli, gdy kula uderza
   // w sciane lub spada na y < 0
}

void RysujZamek()
{
   line(550, 300, 550, 600);
   line(550, 300, 600, 300);
   line(600, 300, 600, 500);
   line(600, 500, 660, 500);
   line(660, 500, 660, 400);
   line(660, 400, 640, 400);
   line(640, 400, 700, 350);
   line(700, 350, 760, 400);
   line(760, 400, 740, 400);
   line(740, 400, 740, 500);
   line(740, 500, 800, 500);
}
```