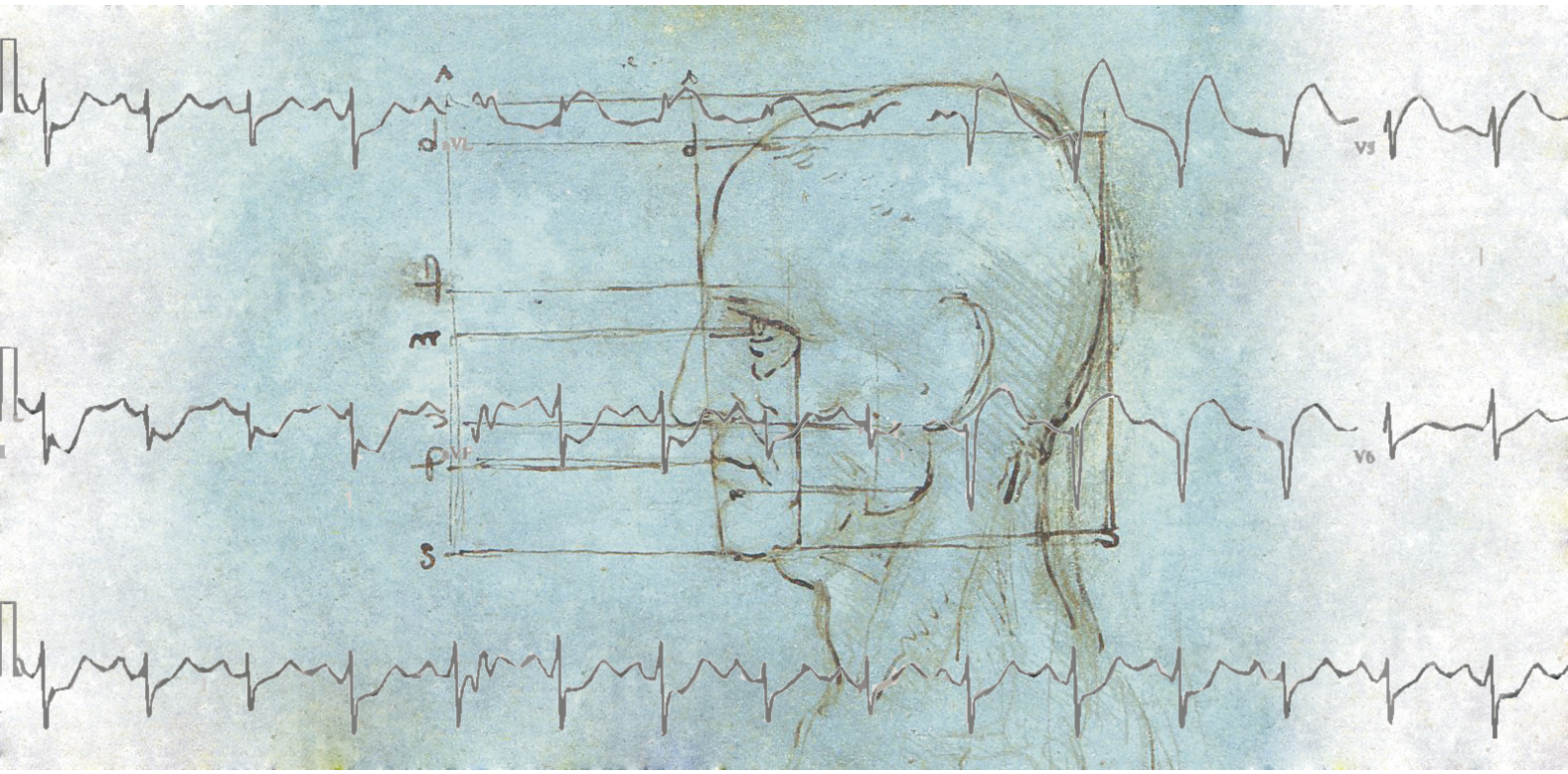


Wstęp do programowania

laboratorium

Wiesław Laskowski

Gdańsk, 2012

Wydział Matematyki, Fizyki i Informatyki
Uniwersytetu Gdańskiego

Skrypt powstał w ramach projektu „Kształcimy profesjonalistów – kompleksowy program kształcenia skierowany do studentów, absolwentów oraz studentów niepełnosprawnych Uniwersytetu Gdańskiego” współfinansowanego przez Unię Europejską w ramach środków Europejskiego Funduszu Społecznego.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIWERSYTET GDAŃSKI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Spis treści

1 Pierwsze kroki w C++ **2**
1 Środowisko pracy . . . 2
2 Pierwszy program . . . 3

2 Dane **5**
1 Typy zmiennych i deklaracje . . . 5
2 Identyfikatory i słowa kluczowe . . . 6

3 Operatory i instrukcje **7**
1 Operatory . . . 7
2 Instrukcje . . . 8
2.1 Instrukcja warunkowa `if` . . . 8
2.2 Pętla `while` . . . 10
2.3 Pętla `do - while` . . . 11
2.4 Pętla `for` . . . 11
2.5 Instrukcja `break` i `continue` . . . 14
2.6 Instrukcja wyboru `switch` . . . 14
2.7 Instrukcja skoku `goto` . . . 16

4 Typy pochodne **17**
1 Tablice . . . 17
2 Wskaźniki . . . 18
3 Dynamiczne alokowanie pamięci . . . 20

5 Funkcje **21**
1 Deklaracja i definicja funkcji. Przeciążanie. . . . 21
2 Wywołanie funkcji . . . 22
3 Czas życia obiektu i obszar ważności nazwy . . . 22
4 Przekazywanie tablic do funkcji . . . 23
5 Standardowa biblioteka funkcji matematycznych . . . 25

6 Programowanie obiektowe **27**
1 Tworzenie obiektów . . . 29
2 Przetwarzanie obiektów . . . 29

7 Operacje wejścia/wyjścia **31**
1 Klawiatura . . . 31
2 Ekran . . . 31
3 Operacje na plikach . . . 33

8 Zadania powtórkowe **37**

9 Rozwiązania wybranych zadań **38**

Rozdział 1

Pierwsze kroki w C++

Czytając skrypt zapoznamy się z podstawami programowania w języku C poszerzonym o wybrane elementy języka C++. Pozwoli to nam na tworzenie prostych jednomodułowych programów w ramach kursu *Wstęp do programowania* dla kierunku *Fizyka medyczna*. Język C++ został zaprojektowany przede wszystkim, jako poszerzenie strukturalnego języka C o mechanizmy obiektowe. W pierwszych rozdziałach zapoznamy się z tymi elementami języków C i C++, które są dla siebie wspólne. Dopiero pod koniec zapoznamy się bardzo pobieżnie z ideą programowania obiektowego.

Materiał zawarty w skrypcie, nie jest kompletnym podręcznikiem dla programistów, a treści kursu będą uzupełniane i rozszerzane w kolejnych latach. Celem na tym etapie, jest zapoznanie z podstawowymi elementami języka oraz wykształcenie pewnych intuicji i nawyków przydatnych w programowaniu. W celu szerszego i głębszego opanowania języka już teraz odsyłam do literatury wymienionej na końcu skryptu.

Rozpoczynamy zatem naszą przygodę z programowaniem. Zaczniemy od krótkiego zaznajomienia się ze środowiskiem, w którym będziemy programować. Napiszemy również nasz pierwszy prosty program i prześledzimy jego działanie. Poznamy w ten sposób elementarny szkielet programu, który będziemy wzbogacać wraz z kolejnymi rozdziałami.

1 Środowisko pracy

Program komputerowy jest sekwencją znaków opisujących obliczenia (zachowanie się komputera). Ta sekwencja znaków podlega pewnym regułom zwanym *językiem programowania*. Działanie programu opisane w sposób zrozumiały dla człowieka nazywamy *kodem źródłowym*, a dla komputera – *kodem maszynowym*. Narzędziem programisty, którego zadaniem jest przetłumaczenie kodu źródłowego języka programowania na język maszynowy, jest *kompilator*.

W toku kursu będziemy używać kompilatora GCC (z ang. GNU Compiler Collection). Naszym środowiskiem pracy będzie aplikacja Dev-C++. Łączy ona między innymi funkcje edytora, w którym tworzymy kod źródłowy i kompilatora. Aplikacja jest bezpłatna i funkcjonuje w oparciu o licencję GNU General Public License. Można ją pobrać ze strony internetowej http://www.bloodshed.net/devcpp.html.

Pliki w języku C++ zapisujemy z następującymi rozszerzeniami:

*.cpp - główny kod źródłowy, w którym znajduje sie funkcja main(), np. 1.cpp,

*.h - pliki nagłówkowe (biblioteki),

*.hpp - definicje klas.

2 Pierwszy program

W tym rozdziale zapoznamy się z podstawową strukturą programu. W tym celu przyjrzyjmy się prostemu programowi.

```
/* Program do przeliczania temperatury
podanej w skali Fahrenheita na temperaturę
w skali Celsjusza */

#include<iostream>

using namespace std;

main(){

float tempFahrenheit;
float tempCelsjusz;

   cout << "Podaj temperature w skali Fahrenheita: ";
   cin >>  tempFahrenheit;

   tempCelsjusz= 5.0/9.0*(tempFahrenheit-32.0); // przelicznik temperatury

   cout << "Temperatura w skali Celsjusza: " <<tempCelsjusz;
}
```

Aby zobaczyć jego działanie uruchamiamy aplikację Dev-C++ i tworzymy pusty dokument (Plik/Nowy/Plik źródłowy). Po wpisaniu kodu źródłowego programu, zapisujemy go pod wybraną przez siebie nazwą (Plik/Zapisz), kompilujemy (Uruchom/Kompiluj) i uruchamiamy (Uruchom/Uruchom).

Każdy pogram w języku C++ musi zawierać funkcję `main` [1]. Od tej funkcji zaczyna się wykonywanie programu. Zawiera ona blok instrukcji pomiędzy nawiasami klamrowymi. W naszym przypadku funkcja `main` rozpoczyna się od zadeklarowania używanych w programie zmiennych `tempFahrenheit` i `tempCelsjusz`. Zmienna `tempFahrenheit` pobierana jest z klawiatury instrukcją `cin`, podczas gdy wartość zmiennej `tempCelsjusz` jest wynikiem działania arytmetycznego zawartego w programie. Program kończy się wyświetleniem jej wartości na ekranie przy użyciu instrukcji `cout`.

Instrukcje wejścia/wyjścia `cout` i `cin` nie są częścią języka C++. Ich definicje zawarte są w jednej z podstawowych *bibliotek*, w które zazwyczaj wyposażone są kompilatory. W tym przypadku jest to biblioteka `iostream`. Aby skorzystać w programie z tej biblioteki musimy na jego początku umieścić linijkę `#include<iostream.h>`, która dołączy do programu *plik nagłówkowy* danej biblioteki. Instrukcja `using namespace std` importuje wszystkie identyfikatory z tzw. *przestrzeni nazw* (w tym wypadku standardowej - `std`). Stosuje się ją po to, aby nie trzeba było przed każdym identyfikatorem z biblioteki standardowej (np. `cout, cin`) pisać `std::`.

Przy okazji wprowadzenia instrukcji `cout` warto wspomnieć o znakach specjalnych. Poniżej zestawionych jest kilka z nich, najczęściej używanych:

`\a` - alarm (sygnał akustyczny terminala)

`\b` - backspace (usuwa poprzedzający znak)

`\f` - wysuniecie strony (np. w drukarce)

`\r` - powrót kursora (karetki) do początku wiersza

[1] Funkcja `main` jest funkcją typu `int` zwracającą wartość 0. Zgodnie z tą uwagą powinna mieć strukturę: `int main(){(...) return 0;}` Większość kompilatorów (w tym nasz) dopuszcza postać funkcji `main(){(...)}`, którą będziemy stosować w skrypcie.

`\n` - znak nowego wiersza

`\''` - cudzysłów

`\'` - apostrof

`\\` - ukośnik wsteczny (backslash)

`\t` - tabulacja pozioma

`\v` - tabulacja pionowa

`\?` - znak zapytania (pytajnik)

Koniec każdej instrukcji musi być oznaczony średnikiem (`;`). Wewnątrz instrukcji można postawić dodatkowe znaki spacji, tabulatory oraz wstawiać dodatkowe puste linie pomiędzy liniami kodu. Są one ignorowane przez kompilator, lecz mogą uczynić tekst źródłowy czytelniejszym. Kompilator ignoruje również *komentarze* – wszystko, co jest zawarte pomiędzy znakami `/*` i `*/` oraz zawartość linijki za znakiem `//`. Pierwszy sposób służy do umieszczania komentarzy blokowych, np. jak na początku naszego programu. Drugi sposób nadaje się do krótkich komentarzy zawieranych w jednej linijce. Czas pracy włożony w tworzenie komentarzy zwraca się z procentem wtedy, gdy zaglądamy do programu po upływie czasu.

W środowisku Dev-C++, jak i w wielu innych, okno zawierające wynik działania programu zamyka się natychmiast po jego zakończeniu. Często uniemożliwia nam to zaobserwowanie efektu programu. Aby utrzymać okno wynikowe otwarte można dodać na końcu programu komendę `system("pause")`. Komenda ta zawiesi działanie programu do czasu naciśnięcia dowolnego klawisza. W tym czasie możemy spokojnie zapoznać się z zawartością okna wynikowego.

Ćwiczenia

1. Napisz najkrótszy możliwy program w C++.
2. Zmodyfikuj omawiany przykładowy program w taki sposób, by przeliczał temperaturę w skali Kelvina na temperaturę w skali Celsjusza.

Rozdział 2

Dane

1 Typy zmiennych i deklaracje

Zmienną nazywany identyfikator (nazwa zmiennej), który związany jest z jakąś wartością. Wartość ta może być modyfikowana w trakcie wykonywania programu. Każda zmienna używana w programie napisanym w języku C musi zostać *zadeklarowana* zanim zostanie użyta. Deklaracja ma postać:

```
typ nazwa_zmiennej;
```

W języku C++ zdefiniowane są następujące podstawowe typy zmiennych:

- *liczby całkowite* (np. 2,-1) – `short int / short`, `int`, `long int / long`
- *znaki alfanumeryczne* (np. '1', 'a') – `char`
- *liczby zmiennoprzecinkowe* (np. 1.23,-4.0, $3.8\text{e}-2 = 3.8 * 10^{-2} = 0.038$) – `float`; *podwójnej precyzji* – `double`, `long double`

Wszystkie powyższe typy mogą występować w dwóch wariantach – ze znakiem lub bez znaku. Uzyskujemy to poprzez poprzedzenie nazwy typu odpowiednio słowem `signed` albo `unsigned`. Domyślnym wariantem, występującym bez użycia powyższych modyfikatorów, jest wariant `signed`.

Oto kilka przykładów zadeklarowania zmiennych:

```
float a;      // zmienna zmiennoprzecinkowa
double b,c;   // łączna deklaracja dwóch zmiennych podwójnej precyzji
int d=2;      // deklaracja wraz z przypisaniem wartości początkowej
```

Każda zmienna zajmuje określoną ilość pamięci, która zależy od typu i czasami od architektury komputera. Operator `sizeof(typ zmiennej)` oblicza rozmiar typu zmiennej w bajtach.

Zamiany typu zmiennej dokonujemy tzw. *operatorem rzutowania*, który ma postać `(nowy_typ)` i występuję przed zmienną. Oto przykład zastosowania operatora rzutowania:

```
float a = 3.14; //zadeklarowanie zmiennej rzeczywistej 'a' i przypisanie jej wartości
int b;          //zadeklarowanie zmiennej całkowitej 'b'

 b = (int)a; // zrzutowanie zmiennej rzeczywistej 'b' na całkowitą

 cout << b;    // wyświetlenie wartości zmiennej b (już całkowitej)
```

Efektem tego programu będzie wyświetlenie liczby 3.

Stałe

Stałą nazywamy identyfikator (nazwa stałej), który związany jest z pewną wartością, która nie ulega zmianie podczas wykonywania programu. Stałe, tak samo, jak zmienne, muszą zostać zadeklarowane w programie przed ich użyciem. Deklaracja stałej przyjmuje postać:

```
const typ nazwa_stałej = wartość_stałej;
```

Oto kilka przykładowych deklaracji stałych:

```
const int k = 1000, M = 1000000;  // kilo, mega
const double h = 6.62606957e-34;  // stała Plancka
```

2 Identyfikatory i słowa kluczowe

Identyfikator, czyli nazwa np. zmiennej lub stałej to ciąg liter i cyfr, rozpoczynający się literą. Dopuszcza się również użycie znaku podkreślenia. Ma znaczenie, czy litera jest mała, czy wielka. Przykładowe różne identyfikatory: `max, Max, max_n , max123`. Dla polepszenia czytelności programu warto używać identyfikatorów związanych z przeznaczeniem. Łatwiej jest wtedy analizować kod takiego programu.

Identyfikatorem nie może być żadne z tak zwanych słów kluczowych, które zostały zestawione w tabeli poniżej:

and	and_eq	asm	auto	bool	break
case	catch	char	class	const	const_cast
continue	default	delete	do	double	dynamic_cast
else	enum	explicit	export	extern	false
float	for	friend	goto	if	inline
int	long	mutable	namespace	new	operator
private	protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast	struct
switch	template	this	throw	true	try
typedef	typeid	typename	union	unsigned	using
virtual	void	volatile	wchar_t	while	

Ćwiczenia

1. Napisz program, który wyświetla rozmiary wszystkich podstawowych typów zmiennych. Przelicz to na zakres wartości, które może przyjmować dany typ zmiennej.
2. Napisz program, który pobiera z klawiatury liczbę rzeczywistą i wyświetla jej część całkowitą i ułamkową.

Rozdział 3

Operatory i instrukcje

W tym rozdziale zapoznamy się z takimi elementami języka C++, jak operatory, wyrażenia i instrukcje.

1 Operatory

Operatorom nie trzeba poświęcać zbyt wiele uwagi, gdyż znaczenie większości z nich zostało poznane już na etapie szkoły podstawowej. Poniżej przyjrzymy się następującym grupom operatorów: arytmetycznym, logicznym i operatorom relacji.

Operatory arytmetyczne i przypisania

Operatory arytmetyczne służą do wykonywania działań na liczbach. W poniższej tabeli zestawione są operatory arytmetyczne wraz z opisem ich działania:

operator	działanie	przykład zastosowania
`+`	dodawanie	`a = b + c`
`-`	odejmowanie lub zmiana znaku	`a = b - c` (odejmowanie), `-x` (zmiana znaku)
`*`	mnożenie	`a = b * c`
`/`	dzielenie	`a = b / c`
`%`	modulo – reszta z dzielenia	`b % 8` – reszta z dzielenia b przez 8
`++`	inkrementacja (zwiększanie o 1)	`i++ (++i)` $\equiv$ `i=i+1`
`--`	dekrementacja (zmniejszanie o 1)	`i-- (--i)` $\equiv$ `i=i-1`

Operatory `++, --` mogą stać zarówno przed, jak i po zmiennej. Jeśli operator stoi przed zmienną, np. `++i` to najpierw jest ona zwiększana o jeden, a dopiero potem ta zwiększona wartość staje się wartością wyrażenia. Jeśli operator stoi za zmienną, np. `i++` to stara wartość zmiennej jest brana jako wartość wyrażenia i dopiero potem jest ona zwiększana o jeden. Przyjrzyjmy się przykładowi:

```
#include<iostream>

using namespace std;

main(){

int m,n,i=2,j=2;

  m = ++i;  // najpierw i=i+1, potem m=i
  n = j++;  // najpierw m=i, potem i=i+1

  cout << m << " " << i << "\n";
  cout << n << " " << j;
}
```

W wyniku jego działania zostaną wyświetlone następujące wartości: $m = 3, i = 3, n = 2, j = 3$.

Operatory przypisania nadają zmiennej po lewej stronie operatora wartość znajdującą się po jego prawej stronie. Operatory przypisania zestawione są w poniższej tabeli:

operator	przykład zastosowania
`=`	`a=1` oznacza przypisanie zmiennej a wartości 1
`+=`	`i+=1` oznacza `i = i + 1`
`-=`	`i-=1` oznacza `i = i - 1`
`*=`	`i*=2` oznacza `i = i * 2`
`/=`	`i/=2` oznacza `i = i / 2`

Relacje

Operatory relacji stosujemy w sytuacjach, gdy konieczne jest porównanie ze sobą dwóch elementów. W poniższej tabeli zestawione są operatory relacji wraz z opisem ich działania:

operator	działanie	przykład zastosowania
`<`	mniejszy niż	`a < b`
`<=`	mniejszy lub równy niż	`a <= b`
`>`	większy niż	`a >b`
`>=`	większy lub równy niż	`a >= b`
`==`	równy	`a == b`
`!=`	różny	`a != b`

Operatory logiczne

Operatory logiczne mają zastosowanie w sytuacjach, gdzie występują różnego rodzaju warunki. W poniższej tabeli zestawione są operatory logiczne wraz z opisem ich działania:

operator	działanie	przykład zastosowania
`&&`	koniunkcja (I, AND)	`(a = b) && (c > d)`
`||`	alternatywa (LUB, OR)	`(a = b) || (a > b)`
`!`	negacja (NIE, NOT)	`a != b`

Ćwiczenia

1. Napisz program, który obliczy pole koła, sfery i objętość kuli o wartości promienia wprowadzonej z klawiatury. Wynik wyświetl w czytelnej postaci.

2 Instrukcje

W tym podrozdziale zapoznamy się z działanie tzw. instrukcji sterujących w języku C++. Ich rolą jest podejmowanie decyzji o przebiegu programu.

2.1 Instrukcja warunkowa `if`

Instrukcja warunkowa `if` służy do uzależnienia wykonania pewnego działania od danego warunku. Ma ona następującą składnię:

```
if (warunek) działanie;     // jeśli warunek jest spełniony wykonaj działanie
```

Jeśli warunek jest spełniony następuje działanie. Poniżej znajdziemy prosty przykład zastosowania tej instrukcji:

```
if (x>0) cout << "x jest wieksze od 0"; // jeśli x > 0 to wyświetl tekst "x jest ..."
```

Instrukcja może być również użyta w bardziej rozbudowanej wersji:

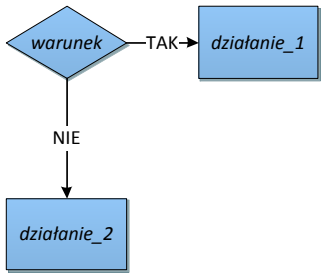
```
if (warunek) dzialanie_1;  // jeśli warunek jest spełniony  wykonaj dzialanie_1
else działanie_2;          // w przeciwnym razie wykonaj działanie_2
```

W tym przypadku jeśli warunek jest spełniony następuje działanie_1, jeśli nie jest spełniony, zostaje wykonane działanie_2.

Jeśli działanie nie jest pojedynczą instrukcją, blok instrukcji należy umieścić w nawiasie klamrowym:

```
if (warunek){
   instrukcja_1;
   instrukcja_2;
   instrukcja_3;
}
else
{
   instrukcja_1;
   instrukcja_2;
}
```

Ta uwaga ma charakter ogólny i będzie używana również w dalszych przypadkach. Graficzna forma instrukcji `if` za pomocą schematu blokowego została przedstawiona na rysunku 3.1.



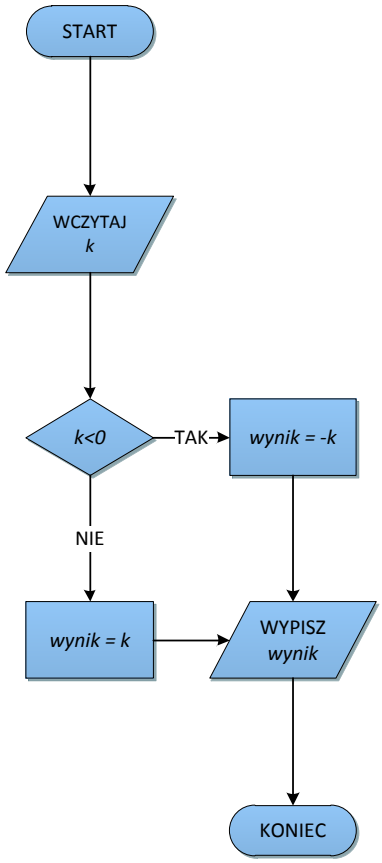
Rysunek 3.1: Schemat blokowy instrukcji `if` – `else`

Przyjrzyjmy się teraz programowi, który będzie realizował funkcję matematyczną $|k|$ tak, aby k było pobierane z klawiatury, a $|k|$ wyświetlane na ekran.

Realizuje go następujący kod programu:

```
#include<iostream>

using namespace std;

main(){
double k, wynik;

  cout << "Podaj liczbe k: ";
  cin >>  k;

  if (k < 0) wynik = -k; // jeśli k>=0 przypisz zmiennej 'wynik' wartość 'k'
  else  wynik = k;  // w przeciwnym wypadku przypisz zmiennej 'wynik' wartość 'k'

  cout << "Wartosc bezwzgledna k=" << wynik;
}
```

Algorytm rozwiązania tego problemu przedstawiony jest na rysunku 3.2.



Rysunek 3.2: Algorytm realizujący funkcję $|x|$.

Ćwiczenia

1. Napisz program, który wyświetla w kolejności rosnącej dwie liczby, które zostały uprzednio wprowadzone z klawiatury.
2. Napisz program, który sprawdza, czy wprowadzona z klawiatury liczba zawiera się w ustalonym przedziale.
3. Napisz program, który dzieli przez siebie dwie liczby wprowadzone z klawiatury. Pamiętaj o tym, że nie można dzielić przez zero.

2.2 Pętla `while`

W trzech kolejnych sekcjach poznamy instrukcje realizujące *pętle*, które pozwalają powtarzać wykonywanie zawartych w nich instrukcji tak długo, dopóki spełniony jest pewien warunek.

Pierwszą z pętli, jest pętla `while`. Jej instrukcja ma następującą składnię:

```
while (warunek) instrukcja;
```

W pierwszej kolejności następuje sprawdzenie, czy warunek jest spełniony. Jeśli tak – wykonywana jest instrukcja. Jeśli nie – następuje wyjście z pętli.

Popatrzmy na następujący przykład:

```
int i=0;

    while (i<5){          // dopóki 'i' jest mniejsze od 5 ...
      cout << i << "\n";  // ... wyświetl 'i'
        i++;                // zwiększ wartość 'i' o jeden
    }
```

W wyniku działania programu zostaną wyświetlone liczby od 0 do 4.

Ćwiczenia

1. Napisz nieskończoną pętlę.
2. Napisz program realizujący prostą grę - *Zgadnij liczbę*. Niech program prosi użytkownika o zgadnięcie uprzednio zdefiniowanej liczby całkowitej z przedziału od 0 do 100. Po podaniu liczby program wskazuje, czy liczba jest mniejsza, większa, czy równa liczbie określonej w programie. Gra kończy się w momencie, gdy użytkownik zgadnie liczbę. Program podaje również liczbę kroków, która była potrzebna do odgadnięcia liczby.

2.3 Pętla `do - while`

Kolejny rodzaj pętli jest bardzo podobny, przy czym najpierw występuje działanie potem dopiero sprawdzany jest warunek i zapada decyzja o kontynuowaniu lub wyjściu z pętli. Uruchamiając program:

```
int i=0;

    do{
        cout << "*";      // wyświetl '*'
        i++;
    }while(i < 0);        // sprawdź, czy i < 0
```

okaże się, że wykona instrukcje pomimo tego, iż warunek $i < 0$ nigdy nie był spełniony. Pętla ma następującą składnię:

```
    do{
        instrukcja;
    }while(warunek);
```

2.4 Pętla `for`

Na trzeci rodzaj pętli należy zwrócić szczególną uwagę, ze względu na jego bardzo częste wykorzystywanie. Tą pętlą jest pętla `for`. Ma ona następującą postać:

```
for ( inicjalizacja; warunek; krok ) blok_instrukcji;
```

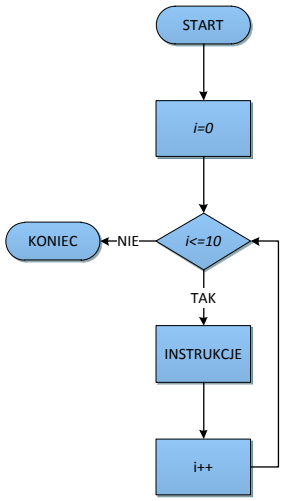
gdzie:

- *inicjalizacja* – to instrukcja wykonywana zanim nastąpi uruchomienie pętli.
- *warunek* – to wyrażenie, którego wartość obliczana jest przed każdym obiegiem pętli. Jeśli jest spełnione – to instrukcje w pętli są wykonywane, jeśli nie – następuje wyjście z pętli.
- *krok* – to instrukcja wykonywana na zakończenie każdego z obiegów pętli.

  Popatrzmy na następujący przykład:

```
#include<iostream>
#include<conio.h>

using namespace std;

main(){

int i;

    for (i=0; i<=10; i++){
      cout << i <<" ";
    }
}
```

Wyrażeniem inicjalizującym pętlę jest `i=0`. Zmienna `i` pełni rolę iteratora pętli. Przypisywana jest jej wartość początkowa 0. Pętla będzie wykonywała zawarte w niej instrukcje tak długo, dopóki `i` będzie mniejsze lub równe 10. Na zakończenie każdego obiegu pętli wartość iteratora `i` zostanie zwiększona o 1. Efektem działania programu będzie wyświetlenie na ekranie liczb od 0 do 10. Schemat blokowy algorytmu pętli przedstawiony jest na rysunku 3.3.

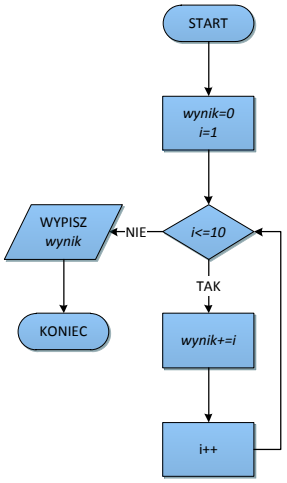


Rysunek 3.3: Algorytm realizujący pętlę `for`.

W tym miejscu warto wspomnieć o jednym, często wykorzystywanym zastosowaniu pętli – implementacji sumowania. Wykorzystamy w tym celu pętlę `for` i operator przypisania `+=`. Popatrzmy na następujący przykład, w którym będziemy chcieli obliczyć sumę wszystkich liczb całkowitych z przedziału od 1 do 10.

```
#include<iostream>
#include<conio.h>

using namespace std;

main(){

int i;          //deklarujemy iterator
int wynik=0;    //deklarujemy zmienną, która przyjmie wartość sumy
                //i nadajemy jej wartość początkową 0

    for (i=1; i<=10; i++){
        wynik += i;  // za każdym obiegiem zwiększamy zmienną 'wynik' o 'i'
    }

   cout << wynik;
}
```

Należy pamiętać, aby zmiennej, której wartość będziemy zwiększać z każdym obiegiem pętli nadać wartość początkową. Gdy o tym zapomnimy, możemy otrzymać nieprzewidywalny wynik. Schemat algorytmy powyższego sumowania przedstawiony jest na rysunku 3.4.



Rysunek 3.4: Suma liczb całkowitych od 1 do 10.

Na koniec tego podrozdziału zwróćmy uwagę na to, iż pętle możemy w sobie zagnieżdżać. Przydatne jest to w sytuacjach, gdy chcemy manipulować więcej, niż jedną zmienną (indeksem). Dla przykładu, chcąc wyświetlić tabelkę składającą się z trzech kolumn, w których w kolejności znajdują się wartości a, b i a/b, dla $a, b, c = \{1, 2, 3\}$, użyjemy podwójnej pętli `for`:

```
for (int a=1; a<=3; a++){  // pętla zewnętrzna zawierająca w sobie
                           // pętlę wewnętrzną

// *** początek pętli wewnętrznej ***
    for  (int b=1; b<=3; b++){
        cout<<a<<"\t"<<b<<"\t"<<a/b<<"\n";
    }
// *** koniec pętli wewnętrznej ***
}
```

Proszę zwrócić uwagę na to, iż w tym przykładzie zmienne a i b zostały zadeklarowane wewnątrz pętli `for`. Jest to możliwe (nie jest możliwe w języku C), ale trzeba pamiętać, iż taka deklaracja będzie obowiązywała tylko wewnątrz danej pętli.

Ćwiczenia

1. Wyświetl na ekranie liczby nieparzyste z zakresu od 1 do 11.
2. Wyświetl na ekranie wszystkie liczby podzielne przez 7 w zakresie od 1 do 100.
3. Zmodyfikuj program z rozdziału 2 tak, aby wyświetlał zestawienie temperatur od $-10°$C do $10°$C (co $1°$C) wraz z odpowiadającymi im temperaturami w skali Fahrenheita.
4. Oblicz następujące wyrażenie: $\sum_{i=0}^{100} i^2$.
5. Oblicz iloczyn wszystkich liczb podzielnych przez 7 w zakresie od 1 do 30.
6. Wyświetl na ekranie tabliczkę mnożenia do 100 (10x10).

2.5 Instrukcja `break` i `continue`

Instrukcja `break` powoduje przerwanie pętli. Jeśli mamy do czynienia z kilkoma zagnieżdżonymi pętlami, instrukcja przerywa tylko wykonywanie tej pętli, na którym poziomie się znajduje, przechodząc o jeden poziom wyżej.

Instrukcja `continue` ma zastosowanie wewnątrz wszystkich pętli. Powoduje ona zaprzestanie wykonywania dalszych instrukcji w danym obiegu pętli. W przeciwieństwie do instrukcji `break`, nie zostaje przerwane działanie pętli. Działanie instrukcji ilustruje program:

```
#include<iostream>
#include<conio.h>

using namespace std;

main()
{

int i;

  for (i=0; i<=10; i++)
  {
    if (i == 5) continue; // jeśli i=5 pomiń wykonywanie dalszych instrukcji w pętli
    cout << i <<" ";
  }
}
```

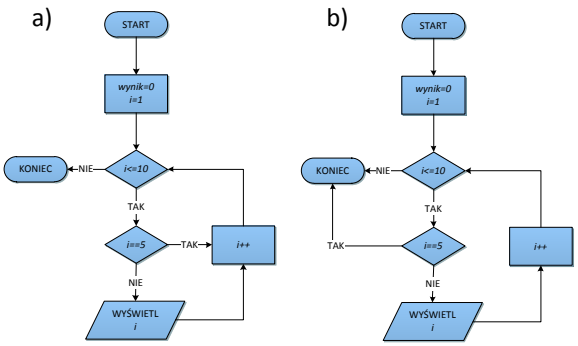
Wynikiem działania programu jest wyświetlenie na ekran wszystkich liczb z przedziału od 0 do 10 za wyjątkiem 5. Różnicę w działaniu pomiędzy instrukcją `continue` i `break` przedstawia rysunek 3.5.

Ćwiczenia

1. Zastąp w powyższym programie instrukcję `continue` instrukcją `break`. Przetestuj jego działanie.
2. Używając jednej pętli `for` i instrukcji `continue`, oblicz następujące wyrażenie: $\sum_{i=0}^{23} i^2 + \sum_{i=46}^{100} i^2$.

2.6 Instrukcja wyboru `switch`

Instrukcja `switch` jest używana w sytuacjach wielokrotnego wyboru. Popatrzmy na następujący fragment programu:



Rysunek 3.5: Różnica w działaniu pomiędzy instrukcją `continue` (a) i `break` (b).

```
// obliczamy wartość 'i' lub np. pobieramy z klawiatury

  switch(i){ // testujemy wartość 'i'
    case 0:                       // jeśli i = 0
      cout << "Zdarzenie A \n";   // wyświetl ten tekst
      break;                      // i opuść instrukcje
    case 1:
      cout << "Zdarzenie B \n";
      break;
    default:                      // jeśli i jest różne od 0 i 1
      cout << "Zdarzenie C \n";   // wyświetl ten tekst
  }
```

W powyższym przykładzie analizujemy wartość zmiennej `i`. Jeśli $i = 0$ to następuje wyświetlenie tekstu *Zdarzenie A* oraz wyście z instrukcji. Podobnie w przypadku $i = 1$. Domyślne działanie instrukcji `switch` określone jest w przypadku *default*. Instrukcje tam zawarte zostaną wykonane wtedy, gdy żaden ze szczegółowych przypadków nie będzie miał miejsca. Brak instrukcji oznaczałby, że zostaną wykonane również wszystkie instrukcje w przypadkach poniżej, czyli dla $i = 0$ zostałyby wyświetlone teksty: *Zdarzenie A*, *Zdarzenie B* i *Zdarzenie C*.

Ćwiczenia

1. Dokończ program z powyższego przykładu. Uruchom i przetestuj. Usuń instrukcje `break`. Uruchom ponownie i przetestuj.

2. Napisz identycznie działający program używając instrukcji `if-else`. Narysuj schemat blokowy algorytmu.

2.7 Instrukcja skoku `goto`

Czasami bywa konieczne dokonanie skoku do konkretnego miejsca w programie. Można to zrealizować instrukcją `goto`. Ma ona postać:

```
goto etykieta;     // program przeskoczy do miejsca w programie oznaczonego etykietą
...
etykieta: instrukcja;  // i od tego miejsca zacznie wykonywanie kolejnych instrukcji
```

Etykietę oznaczamy poprzez użycie identyfikatora (nazwy etykiety) zakończonego dwukropkiem. Użycie instrukcji `goto` w programie powinno być ograniczone do minimum. Powoduje ona, iż możemy utracić kontrolę nad programem. W znacznej większości przypadków zawsze możemy zastąpić ją inną instrukcją. Czasami jednak jest ona pomocna, na przykład przy konieczności wyjścia z bardzo zagnieżdżonych pętli.

Rozdział 4

Typy pochodne

Do tej pory mieliśmy do czynienia z podstawowymi typami danych, takimi jak na przykład `int`, czy `double`. W tym rozdziale zapoznamy się z wybranymi *typami pochodnymi*, które są tworzone na bazie typów podstawowych. Oznacza się je stosując nazwę typu podstawowego i operator deklaracji typu pochodnego.

1 Tablice

Pierwszym typem pochodnym, który poznamy jest *tablica*. Jest ona ciągiem obiektów tego samego typu, które zajmują w pamięci ciągły obszar. Korzyścią z zastosowania tablicy jest możliwość odwołania się po prostu do jej i-tego elementu, bez konieczności definiowania wielu zmiennych. Tablicę definiujemy w następujący sposób:

```
typ nazwa_tablicy[rozmiar_tablicy]
```

Dla przykładu `float a[10]` definiuje tablicę o nazwie `a`, która składa się z dziesięć elementów. Oznacza to, że w pamięci zostało zarezerwowane miejsce na dziesięć liczb typu *float*. Do poszczególnych elementów tablicy odwołujemy się (przypisujemy wartości lub je odczytujemy) poprzez wskazanie konkretnego elementu tablicy, w naszym przypadku np. `a[0]`, `a[1]`, ..., `a[9]`. W tym miejscu trzeba zwrócić uwagę na to, iż elementy tablicy numerowane są od 0. Oznacza to, że jeśli utworzymy tablicę o rozmiarze n, to jej pierwszym elementem jest element o indeksie 0, a ostatnim o indeksie $n-1$.

Przypisanie elementom tablicy wartości może odbyć się na kilka sposobów. Przyjrzyjmy się pewnemu przykładowi. Chcemy, aby nasza tablica, powiedzmy `kierunek`, zawierała wartości współrzędnych wektora $\{2, 1, 5\}$. W tym celu możemy każdemu elementowi tablicy przypisać wartość kolejnej współrzędnej:

```
int kierunek[3];  // deklaracja tablicy o rozmiarze 3
kierunek[0]=2;    //przypisanie wartości poszczególnym elementom tablicy
kierunek[1]=1;
kierunek[2]=5;
```

Można to również zrobić za jednym razem, przypisując wszystkie wartości do tablicy podczas jej deklaracji:

```
int kierunek[3]={2,1,5};
```

W takim przypadku można opuścić rozmiar tablicy w deklaracji, ponieważ kompilator sam obliczy zajmowane miejsce na podstawie ilości podanych elementów. Taka deklaracja wyglądałaby wtedy:

```
int kierunek[]={2,1,5};
```

W powyższych przykładach tablice były jednowymiarowe. Nic nie stoi jednak na przeszkodzie, by tworzyć tablice wielowymiarowe. Przykładowo tablicę dwuwymiarową o rozmiarze 2x3 definiujemy w następujący sposób:

```
int macierz[2][3]={
                    {1, 4, 2},
                    {4, 6, 7}
                   };
```

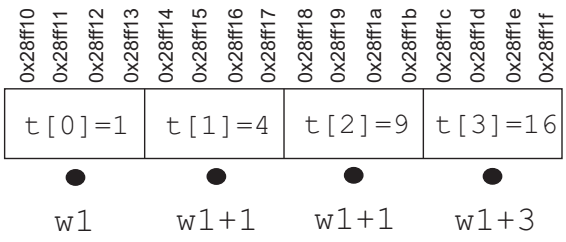
W tym wypadku elementowi tablicy `a[0][0]` została przypisana wartość 1, elementowi `a[0][1]` wartość 4, elementowi `a[1][2]` wartość 7, itd.

Ćwiczenia

1. Napisz program, który zapisze w tablicy wartości funkcji x^2 od 0 do 100 i następnie wyświetli zawartość tabeli.
2. Napisz program, który obliczy sumę wartości elementów tablicy $\{1, 7, 3, 8, 2, 9, 3, 5, 1, 7, 0, 4, 4, 1, 5, 3, 3, 7, 3, 1\}$ oraz znajdzie element maksymalny i minimalny podając ich indeksy.
3. Napisz program, który wykona mnożenie macierzy $\begin{pmatrix} 1 & 5 \\ 6 & 3 \end{pmatrix}$ i $\begin{pmatrix} 2 & 1 \\ 8 & 2 \end{pmatrix}$.
4. Napisz program, który unormuje wektor {1.4,3.8,0.5,4.7}. Do wykonania tego zadania będziesz potrzebował funkcji `sqrt()` (pierwiastek), która zawarta jest w bibliotece `math.h`.
5. Napisz program, który obliczy iloczyn skalarny wektorów {2,7,4,5,1} i {3,7,2,7,4}.
6. Napisz program, który zapisze w tablicy sto pierwszych liczb Fibonacci'ego $f_0 = 0$, $f_1 = 1$, $f_n = f_{n-1} + f_{n-2}$. Wyświetl je wraz z indeksami. Najpierw zadeklaruj tablicę typu `long int`, potem `double`. Porównaj działanie programu.
7. Napisz program, który utworzy dwie tablice `t1` i `t2` o rozmiarach 2x2, a następnie przypisze ich elementom losowe wartości typu `int` z zakresu 1-10. Kolejnym zadaniem programu będzie sprawdzenie, czy tabele zawierają jakieś wspólne elementy (na jakich pozycjach i o jakiej wartości).

2 Wskaźniki

Kolejnym typem pochodnym, z którym się zapoznamy są *wskaźniki*. Najprościej wytłumaczyć ich zastosowania na konkretnym przykładzie. Wyobraźmy sobie sytuację, w której chcemy wyświetlić zawartość czterech kolejnych komórek tablicy, powiedzmy tablicy `t`, przechowującej liczby typu całkowitego (patrz rysunek 4.1). Gdy polecimy wyświetlenie zawartości pierwszej z nich, `t[0]`, zostanie najpierw obliczony adres w pamięci, pod którym się ona znajduje, powiedzmy `0x28ff10`. Następnie zostanie podana wartość znajdująca się pod tym adresem. Przy poleceniu wyświetlenia zawartości kolejnej komórki, `t[1]`, jej adres obliczany jest ponownie. W przypadku wielu operacji, na przykład, na tablicach wielowymiarowych takie działanie pochłania wiele czasu. Można uniknąć tego problemu stosując *wskaźniki*, które pokazują adres, pod którym znajduje się dany obiekt. W przypadku naszego problemu z tablicą wystarczyło ustanowić wskaźnik, powiedzmy `w1`, na adres pierwszej interesującej nas komórki. Następnie należało wyświetlić zawartość komórki znajdującej się pod tym adresem (`0x28ff10`) i pod adresami wskazywanymi przez wskaźnik powiększony o jeden (`0x28ff14`), dwa (`0x28ff18`) i trzy (`0x28ff1c`). Komórki tablicy zajmują w pamięci sąsiadujące miejsca. W tym miejscu odgrywa znaczenie to, że przy deklaracji wskaźnika przypisaliśmy do niego konkretny typ obiektu, na który będzie wskazywał. Wiemy bowiem, że różne typy, na przykład zmiennych, zajmują w pamięci różną ilość miejsca. Oznacza to, że wskaźnik, zadeklarowany jako wskazujący na obiekt typu całkowitego, powiększony o k, nie wskaże komórki pamięci oddalonej o k, a komórkę pamięci oddaloną o k-krotność rozmiaru pamięci, jaki zajmuje zmienna typu całkowitego (w naszym przypadku 4 bity).



Rysunek 4.1: Ulokowanie tablicy `t` w pamięci i wskaźniki.

Popatrzmy na działanie następującego programu:

```
#include<iostream>

using namespace std;

main(){

int i;
int t[4]={1,4,9,16};
int *w1,*w2;       // deklaracja wskaźnika 'w1'('w2')  na obiekt typu 'int'

    w1 = &t[0];    // ustawienie wskaźnika 'w1' na obiekt t[0]
    w2 = &t[1];    // ustawienie wskaźnika 'w1' na obiekt t[1]

    cout << "Adres pamieci pierwszej komorki tablicy: "<< w1 <<"\n";
    cout << "Adres pamieci pierwszej komorki tablicy: "<< w2 <<"\n\n";

    cout << "Pod adresem wskazywanym przez w1 jest: " << *w1 <<"\n";
    cout << "Pod adresem wskazywanym przez w2 jest: " << *w2 <<"\n";
    cout << "Pod adresem wskazywanym przez (w1 + 1) jest: " << *(w1+1) <<"\n";
    cout << "(Pod adresem wskazywanym przez w1) + 1 jest: " << *w1+1 <<"\n";
}
```

W powyższym przykładzie pojawiły się trzy nowe elementy:

- *Deklaracja wskaźnika* – Wskaźnik `w1` wskazujący na obiekt typu całkowitego deklarujemy: `int *w1`. W ogólności jest to `typ_na_ktory_wskazuje_wskaznik *nazwa_wskaznika`.
- *Ustawienie wskaźnika* – Wskaźnik ustawiamy na obiekt za pomocą znaku '`&`'. Jeśli chcemy by wskaźnik `w1` wskazywał na obiekt `t[0]`, to piszemy `w1 = &t[0]`. W ogólności: `nazwa_wskaznika &obiekt_na_ktory_wskazuje`. Po uruchomieniu programu, w pierwszych dwóch linijkach, widzimy wartości wskaźników `w1` i `w2`. Są to po prostu adresy pamięci, w których ulokowane są dane komórki tablicy, dla przykładu `0x28ff10` i `0x28ff14`. Różnica między tymi dwiema liczbami jest rozmiarem typu `int`.
- *Odczytanie adresu wskaźnika* – Gdy chcemy odczytać to, co znajduje się w pamięci komputera pod adresem wskazywanym przez wskaźnik `w1` piszemy: `*w1`. I tak, pod adresem wskazywanym przez wskaźnik `w1`, czyli `0x28ff10` znajduje się wartość przechowywana w komórce tablicy `t[0]`, czyli 1 (trzecia linijka wyniku programu). Pod adresem wskazywanym przez wskaźnik `w2`, czyli `0x28ff14` znajduje się wartość przechowywana w komórce tablicy `t[1]`, czyli 4 (czwarta linijka wyniku programu). W piątej linijce programu poleciliśmy wyświetlić to, co znajduje się w pamięci komputera pod adresem pamięci wskazywanym przez wskaźnik `w1` przesunięty o jeden. W konsekwencji poleciliśmy wyświetlić kolejny element tablicy. Program wiedząc, na jaki typ pokazuje wskaźnik, przesunął go o odpowiednią

liczbę komórek pamięci, w tym przypadku o 4. Aby korzystać z tego inteligentnego zachowania wskaźnika należy pamiętać o nawiasie. W ogólności, w przypadku tablic, `*(w1+k)` oznacza odwołanie się do wartości komórki oddalonej o k od komórki wskazywanej przez wskaźnik `w1`. Zupełnie inny efekt otrzymamy zapominając o nawiasie (6 linijka wyniku programu). W tym przypadku poleciliśmy wyświetlić to, co znajduje się pod adresem przesuniętym o jeden, czyli w naszym przykładzie 0x28ff11. Działanie programu w tym przypadku jest nieprzewidywalne.

Ćwiczenia

1. Wykonaj jeszcze raz ćwiczenia z rozdziału 1. Tym razem użyj wskaźników.

3 Dynamiczne alokowanie pamięci

Często w trakcie pisania programu nie możemy przewidzieć ile będziemy potrzebowali pamięci operacyjnej na działanie naszego programu. Naprzeciw temu problemowi wychodzą operatory `new` i `delete`, których rolą jest odpowiednio przydzielanie i zwalnianie pamięci. Popatrzmy na poniższy przykład, który ilustruje zasadę *dynamicznego przydziału pamięci*.

```
main(){

double *w;        // deklaracja wskaźnika 'w' wskazującego na obiekt typu double

    w = new double [100]; // przydzielenie pamięci dla 100 obiektów typu double
                          // pierwszy obiekt wskazywany jest przez 'w'

    {
      blok instrukcji
    }

    delete[] w; // zwolnienie pamięci
}
```

Proszę zwrócić uwagę, iż przydzielenie pamięci odbywa się przy pomocy wskaźnika, który wskazuje na pierwszy obiekt. Nie nadajemy nazwy obiektom, więc musimy się do nich odwoływać również za pomocą wskaźników. Zwolnienie pamięci w powyższym przykładzie, jak zawsze gdy mamy do czynienia z więcej niż jednym obiektem, odbywa się za pomocą operatora `delete[]`. W przypadku pojedynczego obiektu używamy po prostu `delete`.

Dodatkową zaletą takiego rozwiązania, jest gospodarowanie zasobami pamięci. W przypadku *statycznego przydziału pamięci* `double t[100]`, pamięć przydzielana jest na cały czas działania programu, bez względu na to, czy korzystamy z tablicy `t`, czy nie. Dynamiczne alokowanie pamięci pozwala przydzielić pamięć tuż przed jej użyciem i zwolnić ją, gdy nie będzie już potrzebna.

Rozdział 5

Funkcje

Funkcje w języku C++ należy rozumieć, jako podprogram, który zwraca określoną wartość. Rozbicie programu, na elementy realizujące pewne zadania obliczeniowe poprawia jego czytelność i funkcjonalność. Funkcja jest swego rodzaju *czarną skrzynką*, w której możemy zawrzeć pewne obliczenia, nie zważając później na to, jak je wykonano.

1 Deklaracja i definicja funkcji. Przeciążanie.

Popatrzmy na następujący prosty przykład:

```
float a(float v, float t){ // deklaracja funkcji typu float 'a'
                           // z dwoma argumentami typu float
                           // w nawiasie {} zawarta jest definicja funkcji
    float wynik;

        wynik = v/t;
        return wynik;        // wartość zmiennej 'wynik' zostanie
                             // zwrócona, jako wartość funkcji
}
```

Mamy w nim do czynienia z funkcją, która oblicza przyspieszenie a w ruchu jednostajnie przyspieszonym, gdy w czasie t, prędkość wzrosła o v. Każda funkcja zaczyna się od deklaracji, jakiego typu wartość zwróci. W naszym przypadku `a` jest typu `float`. Przy nazwie funkcji, w nawiasie, podajemy jej argumenty wraz z typami. Nie muszą one być już potem deklarowane wewnątrz funkcji. Poniżej występuje definicja funkcji zamknięta nawiasem klamrowym. Funkcja zwraca wartość występującą po słowie `return`.

Wyjątkiem zasługującym na uwagę jest funkcja typu `void`, która nie zwraca żadnej wartości, a na przykład wypisuje tekst na ekranie lub przetwarza dane. Taka funkcja nie używa `return`. Pełni ona rolę typowego podprogramu, który wykonuje pewne zadanie.

Możliwe jest również ustawienie domyślnych argumentów funkcji. Dla przykładu funkcja

```
void f(int a=1, int b=1){
    cout <<a<<" ,"<<b;
}
```

wywołana w sposób `f()` wyświetli parę liczb (1,1). Przy wywołaniu `f(5)` będzie to para (5,1), natomiast przy pełnym wywołaniu `f(6,4)` – para (6,4). Wartości ustawione, jako domyślne w definicji funkcji ujawnią się, gdy w wywołaniu funkcji pominiemy dany argument.

Deklaracja funkcji musi wystąpić w programie przed jej użyciem. Nie trzeba jej jednak od razu definiować. Definicja może wystąpić później (za funkcją `main`), a nawet w oddzielnym pliku.

Ciekawą i pożyteczną własnością funkcji jest możliwość tak zwanego ich *przeciążania*. Wyobraźmy sobie sytuację, w której deklarujemy w programie dwie funkcje `float potega(float m, int n)` oraz `float potega(float m, float n)`. Pomimo takich samych nazw sytuacja ta jest dozwolona. Kompilator sam dobierze sobie odpowiednią funkcję na podstawie typu danych argumentów.

2 Wywołanie funkcji

Funkcje wywołujemy poprzez napisanie jej nazwy łącznie z nawiasem, w którym znajdują się argumenty funkcji. Przykład wykorzystania funkcji `a` przedstawiony jest poniżej:

```
#include<iostream>

using namespace std;

float a(float v, float t); // deklaracja funkcji 'a'

main(){
 cout << a(2.0,4.0);      // wywołanie funkcji 'a' z argumentami 'v=2.0' i 't=4.0'
}
```

Oczywiście, aby powyższy program zadziałał w pliku musi zostać zawarta definicja funkcji `a(float v, float t)`. Tak, jak zostało zaznaczone w poprzednim podrozdziale, nie musi ona być powiązana z deklaracją i może wystąpić w dowolnym momencie.

Rekurencja

Funkcje w języku C++ mają taką własność, że mogą być wywoływane rekurencyjnie. Oznacza to, iż funkcja może wywołać samą siebie. Popatrzmy na następujący przykład, w którym chcielibyśmy zdefiniować funkcję rekurencyjną zwracającą wartość n-tego wyrazu ciągu danego wzorem $a_n = 5a_{n-1} + 3$, jeśli $a_1 = 5$. Możemy to zrobić w taki sposób:

```
int a(int n) {
   if (n==1) return 5;  // ustalamy element początkowy rekurencji a1=5
                        // to na nim kończy się wykonywanie funkcji

   return 5*a(n-1) +3;  // odwołujemy sie do wartości funkcji w poprzednim kroku
}
```

3 Czas życia obiektu i obszar ważności nazwy

W zależności od sposobu zadeklarowania obiektu możemy określać jego czas życia i obszar ważności nazwy. Możemy to zrobić na kilka sposobów.

- *Globalnie* – Obiekt, który zostanie zadeklarowany na zewnątrz wszystkich funkcji ma zasięg globalny. Dostępny jest wszędzie, we wszystkich funkcjach w obszarze całego pliku.
- *Lokalnie* – Obiekt, który zostanie zadeklarowany wewnątrz jednej funkcji ma zasięg ograniczony tylko do tej funkcji. Po zakończeniu funkcji obiekt zostaje zlikwidowany.
- *Lokalnie - statycznie* – Obiekt lokalny, który zostanie zadeklarowany w taki sposób, po zakończeniu wykonywania funkcji nie przestaje istnieć. Po ponownym wejściu do tej funkcji będzie posiadał swoją ostatnią wartość. Chcąc utworzyć taki obiekt należy dołączyć przed typem w jego deklaracji modyfikator `static`, na przykład `static int w`.

Aby bliżej zapoznać się z tymi własnościami wykonajmy następujący program:

```
#include<iostream>

using namespace std;

void a(); // deklaracja funkcji 'a()'

int t=1;  // zmienna globalna (1)

main()
{
 int t=2; // zmienna lokalna w funkcji 'main' (2)
   cout << "Wartosc 't' w funkcji 'main' (przed): "<<t<<"\n";
   a();
   cout << "Wartosc 't' w funkcji 'main' (po):    "<<t<<"\n";
}

// definicja funkcji a()
void a()
{
  int t=3; // zmienna lokalna w funkcji 'a' (3)
    cout << "Wartosc 't' w funkcji 'a': "<<t;
}
```

Program polega na tym, iż najpierw z poziomu funkcji `main` wyświetla aktualną wartość zmiennej `t`, następnie uruchamia funkcję `a()`, której zadaniem jest również wyświetlenie wartości `t` i znów wyświetla wartość zmiennej z poziomu funkcji `main`. Zmienna została zadeklarowana w trzech różnych miejscach. Przed skompilowaniem programu zakomentujmy linijkę (1). Po uruchomieniu okazuje się, że program wyświetlił wartości zmiennej `t` w sposób niezależny. Pomimo takiego samego identyfikatora, zmienna zdeklarowana w funkcji `a()` nie ma wpływu na zmienną w funkcji `main`. dodatkowe zakomentowanie (2) lub (3) powoduje wskazanie błędu przez kompilator. Żadna deklaracja nie ma bowiem zasięgu wykraczającego poza funkcje, gdzie została umieszczona. Sytuacja zmieni się, gdy odkomentujemy (1). Wtedy po zakomentowaniu (2) i (3) program nadal się uruchamia. Wartość zmiennej `t` w każdym miejscu jest taka sama. Ma ona w tej chwili zasięg globalny. W przypadku uruchomienia programu z wszystkimi odkomentowanymi deklaracjami wynik działania programu jest taki, jak w przypadku zakomentowania (1). Lokalne deklaracje mają w swoim obszarze priorytet wobec deklaracji globalnej.

4 Przekazywanie tablic do funkcji

Do tej pory zajmowaliśmy się przypadkami, w których argumentami funkcji były zmienne. Czasami jednak potrzebujemy stworzyć funkcję, która operje na tablicach. W takim przypadku jej deklaracja wygląda następująco

```
typ nazwa_funkcji(typ_tablicy nazwa_tablicy[])
```

Do funkcji przekazywany jest adres zerowego elementu tablicy związanej z nazwą `nazwa_tablicy`. Popatrzmy na przykład, w którym mamy do czynienia z funkcją obliczającą normę wektora o określonej długości.

```
double norm(int size, double t[]) // deklaracja funkcji dwóch parametrów:
{                                 //        size - długość wektora
                                  //        t[]  - jednowymiarowa tablica

double temp=0;

   for(int i=0; i<size; i++)      // obliczanie normy wektora
   {
      temp += t[i]*t[i];
   }

   return sqrt(temp);            // zwrócenie wartości normy przez funkcję
}
```

Mając już funkcję obliczającą normę wektora zadanego w formie tablicy, możemy również stworzyć funkcję normującą taki wektor. Wyglądałaby ona następująco:

```
void normed(int size, double t[]) // deklaracja funkcji (taka sama, jak poprzednio)
{

   double norma = norm(size, t);  // obliczenie normy wektora korzystając z
                                  // poprzedniej funkcji
   for(int i=0; i<size; i++)
   {
      t[i]=t[i]/norma;            // przeskalowanie współrzędnych wektora
   }

}
```

Proszę zauważyć, iż powyższa funkcja nie jest typu `void` – nie zwraca, żadnej wartości, a jedynie dokonuje operacji na tablicy. Działanie obu funkcji możemy sprawdzić uruchamiając program:

```
#include<iostream>
#include<math.h>

using namespace std;

main(){
double wektor[] = {1.0,1.0,1.0};

   cout<<t[0]<<"\t"<<t[1]<<"\t"<<t[2]<<"\n"; //wyświetlenie oryginalnych wartości
                                             //współrzędnych wektora

   normed(3,wektor);                         //unormowanie wektora

   cout<<t[0]<<"\t"<<t[1]<<"\t"<<t[2];       //wyświetlenie wartości współrzędnych
                                             //unormowanego wektora
}
```

W przypadku przekazywania do funkcji tablic dwuwymiarowych, musimy w deklaracji funkcji podać liczbę kolumn tablicy, na przykład `double tablica[][16]`. Przykładowa deklaracja funkcji, której argumentem jest tablica trójwymiarowa może mieć postać `double tablica[][12][15]`. W tablicach wyielowymiarowych jesteśmy zobowiązani podać rozmiar tablicy pomijając lewą skrajną wartość.

5 Standardowa biblioteka funkcji matematycznych

Standardowa biblioteka funkcji matematycznych ma plik nagłówkowy `math.h`. Zawiera w sobie definicje następujących funkcji:

`double abs(double d)`	$\lvert d \rvert$
`double ceil(double d)`	$\lceil d \rceil$
`double floor(double d)`	$\lfloor d \rfloor$
`double sqrt(double d)`	$\sqrt{d}$
`double pow(double d, double e)`	d^e
`double pow(double d, int i)`	d^i
`double cos(double d)`	$\cos d$
`double sin(double d)`	$\sin d$
`double tan(double d)`	$\tan d$
`double acos(double d)`	$\arccos d$
`double asin(double d)`	$\arcsin d$
`double atan(double d)`	$\arctan d$
`double atan2(double x, double y)`	$\arctan(x/y)$
`double cosh(double d)`	$\cosh d$
`double sinh(double d)`	$\sinh d$
`double tanh(double d)`	$\tanh d$
`double acosh(double d)`	arccosh d
`double asinh(double d)`	asinh d
`double atanh(double d)`	atanh d
`double exp(double d)`	e^d
`double log(double d)`	$\log d$
`double log2(double d)`	$\log_{10} d$

Biblioteka zawiera również definicje stałych, na przykład: $\pi \equiv$ `M_PI`, $\pi/2 \equiv$ `M_PI_2`, $\pi/4 \equiv$ `M_PI_4`, $e \equiv$ `M_E`.

Generowanie liczb pseudolosowych

Warto w tym miejscu wspomnieć o pewnej matematycznej funkcji `rand()`, która generuje liczby pseudolosowe z przedziału od 0 do `RAND_MAX`. Program, który generuje liczby pseudolosowe od 0 do 9 może mieć postać:

```
#include<iostream>
#include<stdlib.h>
#include<time.h>

main(){
int k;
   srand(time(0));
   k = int( ( (double)rand()/RAND_MAX ) *9 );
}
```

Ćwiczenia

1. Przekształć program z rozdziału 2 tak, aby obliczenia były dokonywane przez funkcje.
2. Napisz funkcję $sign(x) = \begin{cases} -1 & \text{dla } x < 0 \\ 0 & \text{dla } x = 0 \\ 1 & \text{dla } x > 0 \end{cases}$
3. Napisz funkcje `potega`, która obliczy wartość m^n.

4. Napisz program, który pobiera trzy liczby rzeczywiste: a, b oraz c i następnie oblicza rzeczywiste (zespolone) pierwiastki równania kwadratowego $ax^2 + bx + c = 0$.
5. Napisz funkcję, która oblicza kombinację k elementów z n. Funkcję *silnia* zdefiniuj, jako odrębną funkcję (najpierw użyj pętli `for`, a potem zdefiniuj funkcję, jako rekurencyjną).
6. Napisz funkcje, która podaje wartość n-tej liczby Fibonacci'ego, które dane są wzorami: $f_0 = 0$, $f_1 = 1$, $f_n = f_{n-1} + f_{n-2}$.
7. Napisz funkcję, która oblicza ilość cyfr w liczbie całkowitej wprowadzonej z klawiatury.
8. Napisz funkcję, która oblicza wyznacznik zadanej macierzy 2x2.
9. Napisz funkcję, która oblicza iloczyn skalarny dwóch zadanych wektorów o dowolnym wymiarze.
10. Napisz funkcję, która wymnoży dwie macierze o wymiarze 5x5.
11. Napisz funkcje, która wyświetli tablicę dwuwmiarową o rozmiarze 3x3.

Rozdział 6

Programowanie obiektowe

W typ rozdziale zajmiemy się esencją języka C++, czyli programowaniem obiektowym. Należy jednak zwrócić uwagę na to, iż tylko otrzemy się o ten temat przedstawiając ogólne zasady i możliwości. Osoby chcące zgłębić ten temat mogą odnaleźć więcej informacji w literaturze podanej na końcu skryptu.

Język C++ pozwala na definiowanie własnych typów danych. Do tej pory, jeśli pisaliśmy program przeliczający temperatury lub program obliczający pole powierzchni koła, używaliśmy tego samego typu danych np. `float`. Kompilator nie dbał o to, czy przeliczając temperaturę omyłkowo nie podstawiliśmy zmiennej potrzebnej do obliczenia pola koła i odwrotnie. Liczyło się tylko to, by była ona typu `float`. W języku C++ możemy tworzyć nowe struktury danych (klasy obiektów), z którymi powiązane są metody ich przetwarzania. Takie powiązanie funkcji z danymi sprawia, iż kompilator nie dopuści do wysłania argumentu typu *temperatura*, zamiast argumentu typu *promień*, pomimo tego, iż oba są typu `float`.

Dobrą ilustracją idei programowania obiektowego jest system Microsoft Windows. Gdy popatrzymy na pulpit, zobaczymy ikony różnych plików – powiemy obiektów klasy "pliki". Gdy najedziemy na ikonę pliku wskaźnikiem myszy i naciśniemy jej prawy klawisz, ukaże się menu – powiemy wykaz funkcji składowych. Co więcej, okaże się, iż plik muzyczny będzie miał inne funkcje składowe, niż plik tekstowy. Szybko zauważymy, iż wszystkie pliki muzyczne (a napewno pliki muzyczne jednego rodzaju) będą miały ten sam zestaw funkcji. Od razu nasunie się nam spostrzeżenie, iż jako obiekty należą do klasy "pliki muzyczne", która się nieco różni od klasy "pliki tekstowe". Niemożliwe jest wydrukowanie pliku muzycznego, czy odtworzenie pliku tekstowego. Takie funkcje nie przynależą do obiektów z tych klas.

W celu poznania elementarnych własności programowania obiektowego prześledźmy działanie następującego programu:

```
#include<iostream>
#include<math.h>

using namespace std;

class zespolona{

    // D A N E

      private:    // wszystkie elementy składowe klasy zadeklarowane poniżej
        double x; // są dostępne tylko dla funkcji składowych z tej klasy
        double y; // (metod) lub funkcji zaprzyjaźnionych
```

```
      public:     // wszystkie składowe klasy zadeklarowane poniżej są dostępne
                  // dla wszystkich funkcji
        zespolona(double a, double b){ // konstruktor obiektu

          x = a;
          y = b;
        }

    // M E T O D Y    // deklaracje metod (funkcji składowych)

        void dodaj(zespolona p){  //dodawanie liczb zespolonych
          x += p.x;
          y += p.y;
        }

        void wyswietl() const;  //wyświetlanie liczby zespolonej
        double re() const;   //część rzeczywista liczby zespolonej
        double im() const;   //część urojona liczby zespolonej
        friend double modul(zespolona); // deklaracja funkcji zaprzyjaźnionej
                                        // moduł liczby zespolonej
      };

   // D E F I N I C J E   M E T O D  (funkcji składowych)


void zespolona::wyswietl() const{
 if (y>=0) cout <<  x << "+" << y << "i";
 else cout <<  x  << y << "i";
}

double zespolona::re() const{
 return x;
}

double zespolona::im() const{
 return y;
}

double modul(zespolona p){
 return sqrt(p.x * p.x + p.y * p.y);
}


main(){
 zespolona a(1,1);  // tworzymy obiekt o nazwie 'a' (liczbę '1 + i')
 zespolona b(2,-3); // tworzymy obiekt o nazwie 'b' (liczbę '2 - 3i')

 cout<<"a= "<<a.wyswietl(); cout<<"\n";  //wyświetlamy obie liczby
 cout<<"b= "<<b.wyswietl(); cout<<"\n";
 cout<< "|a|= " << modul(a) << "\n";    //obliczamy ich moduły
 cout<< "|b|= " << modul(b) <<"\n\n";

 a.dodaj(b); //dodajemy do liczby 'a' liczbę 'b' (od tej pory 'a' jest nową liczbą)

 cout<< "a'=" << a.wyswietl(); cout<<"\n"; // wyświetlnamy nową liczbę 'a'
 cout<< "\n" << "Re(a)= "<<a.re(); // jej część rzeczywistą
 cout<< "\n" << "Im(a)= "<<a.im(); // i urojoną
}
```

Analizę powyższego programu zaczniemy od wyjaśnienia pojęcia *klasy*. Klasa jest właśnie zdefiniowanym przez nas nowym typem, który może zawierać dane i funkcje. Określa ona, jak wyglądają wszystkie obiekty, które do niej należą i jak możemy z nimi pracować. Deklaracja klasy wygląda w sposób następujący

```
class nazwa_klasy{
...
}
```

Wewnątrz ciała klasy następują deklaracje danych i funkcji składowych danej klasy. W naszym przypadku, klasa reprezentuje obiekty będące liczbami zespolonymi. Zawiera ona:

- *dane*: dwa elementy obiektu (pola) `x` i `y`, które odpowiadają części rzeczywistej i urojonej danej liczby zespolonej (obydwa są typu `double`),
- *funkcje*: `zespolona`, `dodaj`, `re`, `im`, `wyswietl`, które służą do przetwarzania obiektów z tej klasy,
- *przyjaciela*: `modul`, funkcje nie będącą składnikiem klasy, ale mającą dostęp do jej obiektów.

Wszystkie składniki klasy podzielone są ze względu na uprawnienia dostępu na dwa rodzaje: *public* i *private*. Do tych drugich mają dostęp jedynie funkcje składowe danej klasy i przyjaciele. Pierwsze są dostępne dla wszystkich funkcji.

1 Tworzenie obiektów

Specjalną funkcją, której zadaniem jest tworzenie obiektów należących do danej klasy jest *konstruktor*. Poznać można go po tym, iż posiada taką samą nazwę, jak klasa. Wywołanie konstruktora

```
nazwa_klasy nazwa_obiektu (wartosci_elementów/pól);
```

utworzy nowy obiekt. I tak, `zespolona a(1,5)` utworzy obiekt `a` o polach równych odpowiednio 1 i 5. Taki obiekt będzie reprezentował liczbę zespoloną $1 + 5i$.

Warto wspomnieć, iż poza konstruktorami istnieją również *destruktory*, których zadaniem jest likwidowanie obiektów. Mają one jednak zastosowanie, gdy mamy do czynienia z dynamicznym przydzielaniem pamięci do obiektu (operator `new`).

2 Przetwarzanie obiektów

Aby odnieść się do konkretnego składnika obiektu (w naszym przypadku `x` – oznaczający część rzeczywistą lub `y` – oznaczający część urojoną) stosujemy operator ".", to jest

```
obiekt.skladnik
```

Jeśli popatrzymy na definicję funkcji `modul` to zobaczymy, iż jest ona funkcją kwadratów obu składników naszej klasy, do których odnosimy się poprzez `p.x` lub `p.y`, gdzie `p` jest obiektem, który przekazujemy jako argument do funkcji.

Do pracy z obiektami służą funkcje. Aby funkcja miała dostęp do obiektów danej klasy musi zostać w niej (w jej ciele) zadeklarowana. Definicja funkcji może mieć miejsce jednocześnie z deklaracją (tak, jak w przypadku funkcji `dodaj`) lub poza ciałem klasy (jak w przypadku pozostałych funkcji). Gdy chcemy zdefiniować funkcje poza ciałem klasy musimy poprzedzić jej nazwę nazwą klasy i operatorem "::", np. `zespolona::wyswietl`. Funkcje oznaczone modyfikatorem `const` nie mogą modyfikować obiektu, a jedynie odczytywać jego dane. Proszę zauważyć, iż są to funkcje bezargumentowe (do odczytania składowych obiektu nie potrzebujemy przekazania go, jako argumentu funkcji).

Specjalną kategorią funkcji są *przyjaciele*. Deklaracja takiej funkcji poprzedzona jest modyfikatorem `friend`. Oznacza to, że taka funkcja, pomimo, iż nie jest składową danej klasy możę uzyskać dostęp do

prywatnych składników klasy. Przyjaciel może być funkcją składową innej klasy lub po prostu zwykłą zewnętrzną funkcją i być wykorzystywany przez obiekty należące do innych klas. W naszym przykładzie taką funkcją jest `modul`, który jednocześnie nadaje się na przykład do obliczania normy wektora i mógłby być składową lub przyjacielem klasy wektorów.

Funkcje składowe wywoływane są używając operatora ".". Dla przykładu `a.wyswietl()`, uruchomi funkcje `wyswietl` na obiekcie `a`. Przyjaciele są wywoływani jak zwykłe funkcje, np. `modul(a)`. Przykłady wywoływania funkcji możemy zobaczyć w funkcji `main` przykładowego programu.

Na zakończenie przyjrzyjmy się sytuacji, w której utworzymy następującą funkcję:

```
void function(zespolona p){
 cout <<  p.x
}
```

Nie jest ona, ani funkcją składową, ani zaprzyjaźnioną. Uzupełnijmy kod naszego programu o powyższą funkcję i spróbujmy uruchomić kompilator. W odpowiedzi uzyskamy komunikat "double zespolona::x' is private". Tak kończy się uruchomienie funkcji, która nie posiada uprawnień do operowania na obiektach klasy.

Ćwiczenia

1. Napisz funkcję składową klasy `zespolona`, która sprzęga daną liczbę zespoloną.
2. Zaprojektuj funkcje składowe klasy `zespolona`, która mnoży i dzieli przez siebie dwie liczby zespolone (dwa obiekty z tej klasy). Zrób to na kilka sposobów: a) pierwszy obiekt po operacji przyjmuje wartość iloczynu (ilorazu), b) wyświetlana jest wartość iloczynu (ilorazu), c) tworzony jest nowy obiekt będący iloczynem (ilorazem).
3. Utwórz klasę, której obiektami są punkty na płaszczyźnie trójwymiarowej. Zaprojektuj funkcje: tworzącą obiekty (konstruktor), przesuwającą punkt o zadany wektor, obliczającą odległość dwóch punktów od siebie oraz punktu od początku układu współrzędnych.

Rozdział 7

Operacje wejścia/wyjścia

W tym rozdziale zajmiemy się operacjami wejścia/wyjścia, czyli takimi, które umożliwiają komunikowanie się komputera ze środowiskiem. Zaczniemy od komunikacji z użytkownikiem za pomocą ekranu (wyjście) i klawiatury (wejście). Z tymi operacjami zaznajomiliśmy się już w poprzednich sekcjach. Następnie przejdziemy do umówienia operacji na plikach (weście i wyjście). Operacje wejścia/wyjścia nie są częścią samego języka C++. Ich wykonywanie odbywa się dzięki bibliotekom, w szczególności bibliotece `iostream` zalecanej do stosowania w programowaniu w C++. Wszystkie operacje będziemy rozważać na bardzo podstawowym poziomie.

1 Klawiatura

Wprowadzanie danych należy rozumieć, jako strumień bitów płynących w tym przypadku z klawiatury do programu. Takie strumień oznaczony jest poprzez znane nam już `cin`. W tym miejscu potrzebujemy również operatora, który dokona odczytania informacji ze strumienia. Jest nim operator `>>` (symbolicznie oznacza ruch). Zatem zapis:

```
cin >> x;
```

oznacza przepływ informacji ze strumienia `cin` (klawiatura) do zmiennej x. W praktyce oznacza to pobranie informacji z klawiatury i przypisanie jej do zmiennej x. W przypadku, gdy chcemy wczytać z klawiatury więcej, niż jedną zmienną można to zrobić na dwa sposoby. Pierwszy z nich polega na wielokrotnym powtórzeniu poleceń:

```
cin >> x;
cin >> y;
```

W tym przypadku jeśli chcemy przypisać zmiennym x i y odpowiednio wartości 5.20 i 7.13, należy to zrobić: 5.20[enter]7.13 . Drugi sposób prowadzi do bardziej zwięzłego zapisu:

```
cin >> x >> y;
```

Tym razem wartości wprowadzamy za jednym razem oddzielając je spacją: 5.20[spacja]7.13.

2 Ekran

W przypadku ekranu strumień informacji jest reprezentowany poprzez również znane nam `cout`, a operator przekazywania do strumienia to `<<`. Poniżej widzimy kilka możliwych "konfiguracji" tej operacji:

```
cout << x;            // wyświetlnenie zmiennej 'x'
cout << "Witaj";      // wyświetlenie tekstu
cout << "x= " << x;   // wyświetlenie tekstu i zmiennej
```

W tym miejscu warto omówić kilka z tzw. manipulatorów, które mogą być nam przydatne, przy formatowaniu wyświetlanych wyników. Wymagają one użycia biblioteki `iomanip.h`.

- `setw` — *set width*, czyli ustaw szerokość. Dzięki użyciu manipulatora `setw(m)` możemy sprawić, że liczba n cyfrowa zostanie wyświetlona za pomocą $\max(m, n)$ znaków. Pozostałe znaki zostaną wypełnione domyślnie spacjami. Przykład użycia tego manipulatora znajduje się poniżej:

```
#include<iostream>
#include<iomanip>

using namespace std;

main(){
    cout<<setw(4)<<1025<<" PLN \n";
    cout<<setw(4)<<4<<" PLN \n";
    cout<<setw(4)<<52<<" PLN \n";
    cout<<setw(4)<<162<<" PLN \n";
}
```

  Wynikiem jego działania będzie słupek wyrównany do prawej strony:

```
1025 PLN
   4 PLN
  52 PLN
 162 PLN
```

- `setfill` — dzięki niemu można określić, jakimi znakami zostaną dopełnione liczby, których liczba cyfr jest mniejsza, niż liczba określona manipulatorem `setw`. Zamiana pierwszej linii w powyższym przykładzie na następującą:

```
cout<<setw(4)<<setfill('.')<<1025<<" PLN \n";
```

  spowoduje, iż efektem działania programu będzie kolumna:

```
1025 PLN
...4 PLN
..52 PLN
.162 PLN
```

- `setprecision` — bardzo istotny manipulator, dzięki któremu może określi z jaką dokładnością ma być wyświetlona liczba. Domyślną dokładnością jest sześć miejsc po przecinku. Argument użyty w manipulatorze określa liczbę cyfr liczby, które mają zostać wyświetlone (wszystkie liczby, zarówno przed, jak i po przecinku). Dla zilustrowania można użyć przykładu:

```
#include<iostream>
#include<iomanip>

using namespace std;

main(){
    for (int i=1; i<=10; i++){
        cout<<setprecision(i)<<sqrt(3.0)<<"\n";
    }
}
```

Wynikiem działania programu będzie kolumna liczb:

2
1.7
1.73
1.732
1.7320
1.73205
1.732050
1.7320508
1.73205080
1.732050808

- `setiosflags` — ten manipulator ustawia tzw. flagę. My zwrócimy uwagę tylko na jedną z nich, a mianowicie na flagę `scientific`. Dzięki niej liczba zostanie wyświetlona w konwencji naukowej, czyli na przykład 5.315421e+002 $\equiv 5.315421 \cdot 10^2 = 531.5421$. Zastosowanie manipulatora przedstawia poniższa linia kodu:

```
cout<<setiosflags(ios::scientific)<<1000.0;    // wyświetli liczbe
                                               // w postaci 1.000000e+003
```

Warto w tym rozdziale wspomnieć, iż dane przed wyświetleniem na ekranie przechowywane są w buforze i uwalniane dopiero po nagromadzeniu się. Skraca to czas działania programu ze względu na ograniczenie czasochłonnej komunikacji. Czasami jednak może zależeć nam na wymuszeniu opróżnienia bufora i wyświetleniu jego zawartości na ekranie. Służy do tego manipulator `flush`. Należy go użyć w następujący sposób:

```
cout << "Tekst do wyswietlenia natychmiast" << flush;
```

Podobny efekt możemy uzyskać używając manipulatora końca linii `endl`, którego działanie jest identyczne, jak `\n` i dodatkowo wymusza opróżnienie bufora.

3 Operacje na plikach

Aby móc czytać zawartość plików i zapisywać do nich potrzebujemy dołączyć plik nagłówkowy `fstream.h`. Umożliwi on nam posłużenie się klasami, które zapewnią nam następujące operacje:

- ofstream — zapis do plików,
- ifstream — czytanie plików,
- fstream — zapis i czytanie plików.

Aby czytać z pliku lub do niego zapisywać trzeba:

1. Zdefiniować strumień, czyli wykreować obiekty powyższych klas.
2. Określić strumieniowi z jakim plikiem będzie się komunikował i otworzyć go.
3. Wykonać operacje wejścia/wyjścia.
4. Zlikwidować strumień.

Zapis do pliku

Spróbujmy zatem dokonać prostego zapisu do pliku według powyższej procedury. Realizuje do następujący program:

```
#include<iostream>
#include<fstream>

using namespace std;

main(){
ofstream przyklad;  // utworzenie strumienia o nazwie 'przyklad'

 przyklad.open("1.txt"); // otwarcie pliku o nazwie '1.txt'
                         // związanego ze strumieniem 'przyklad'
 przyklad << "Wstep do programowania"; // przekazanie tekstu do strumienia
                                       // (pliku 'przyklad')
 przyklad.close(); //zamkniecie strumienia
}
```

Efektem działania programu będzie utworzenie pliku o nazwie "1.txt" o zawartości "Wstep do programowania". W tym miejscu należy wspomnieć o tym, iż plik może być otwierany w różnych trybach. Ustawia się je w funkcji `open`. Są nimi:

- `in` — otwarcie pliku do odczytu (domyślny w klasie `ifstream`),
- `out` — otwarcie pliku do zapisu (domyślny w klasie `ofstream`),
- `app` — otwarcie pliku do zapisu (informacje będą dopisywane do pliku)
- `trunc` — otwarcie pliku do zapisu (skasowanie starej treści pliku, domyślnie w trybie `out`)
- `nocreate` — otwarcie pliku, gdy już istnieje,
- `noreplace` — otwarcie pliku, tylko gdy nie istnieje.

W powyższym przykładzie nie został ustawiony żaden tryb, więc domyślnie został przyjęty tryb dla klasy `ofstream`, czyli `out`, a dokładniej jego wersja `trunc`. Popatrzmy co się stanie, jeśli linię, w której otwieramy plik zastąpimy poprzez:

```
przyklad.open("1.txt", ios::app);
```

Po kilkukrotnym uruchomieniu programu okaże się, iż w pliku "1.txt" występuje wiele razy napis "Wstep do programowania". Stało się tak dlatego, iż w trybie `app` nowe dane dopisywane są do starego pliku bez wymazywania jego zawartości. Na koniec warto wspomnieć, iż łączenie trybów odbywa się za pomocą znaku "|", np. polecenie `open("1.txt", ios::out | ios::nocreate)` oznacza otwarcie pliku w trybie `out` tylko wtedy, gdy plik już istnieje.

Zapisywanie danych do pliku jest ważną umiejętnością. Przydaje się szczególnie w sytuacji, gdy wynik działania programu nie jest pojedynczą liczbą, którą możemy przepisać z ekranu monitora, a całym zbiorem danych. Przyjrzyjmy się modyfikacji pierwszego naszego programu z rozdziału 2.

```
/* Program do przeliczania temperatury
podanej w skali Fahrenhaita na temperature
w skali Celsjusza */

#include<iostream>
#include<fstream>

using namespace std;

double fharenheitToCelsjusz(double tempFahrenheit){ // funkcja przeliczająca
  return 5.0/9.0*(tempFahrenheit-32.0);             // temperaturę
}
```

```
main(){

ofstream file;

   file.open("fahrenheit.txt");      // otwarcie pliku w domyślnym trybie zapisu

   file << "T [F] \t" << "T [C]\n"; // zapis do pliku

   for (double i=0.0; i<=100.0; i++){
      file << i << "\t" <<fahrenheitToCelsjusz(i) << "\n"; // zapis do pliku
   }

   file.close();  // zamknięcie pliku

   cout << "Zapisano do pliku!";  // wyświetlenie komunikatu
}
```

Po uruchomieniu programu powinien pojawić się plik o nazwie *fahrenheit.txt* zawierający zestawienie temperatur.

Odczytywanie zawartości pliku

Przyjrzyjmy się teraz odwrotnemu problemowi, a mianowicie wczytywaniu danych zapisanych uprzednio w pliku. Utwórzmy najpierw plik tekstowy w dowolnym edytorze o dowolnej treści i nazwijmy go *przyklad.dat*. Umieśćmy go w tym samym katalogu co nasz program. Spróbujmy teraz odczytać zawartość programu i wyświetlić go na ekranie. W tym celu sprawdźmy działanie programu:

```
#include<iostream>
#include<fstream>

using namespace std;

main(){
 char c;   // deklarujemy zmienną znakową 'c'

  ifstream input("przyklad.dat"); // deklarujemy strumień z pliku 'przyklad.dat'

 cout<<"ZAWARTOSC PLIKU:";
 cout<<"\n------------------------------\n\n";

  while(input.get(c)){   // odczytujemy w pętli po kolei wszystkie znaki w pliku
    cout<<c;             // i przypisujemy je zmiennej 'c'
  }                      // wyświetlamy wartość zmiennej 'c'

 cout<<"\n \n------------------------------";
}
```

W tym miejscu należy zwrócić uwagę na funkcję odczytującą zawartość pliku. Jest nią `get(c)`. Powoduje ona pobranie pojedynczego znaku ze strumienia `input` (w tym przypadku płynącego z pliku) i przypisanie go do zmiennej `c`. Inną pożyteczną funkcją jest `int get()`. W odróżnieniu do poprzedniej, wczytany znak, jest zwracany przez funkcje, jako `int`, reprezentując kod ASCII znaku. Można go potem zrzutować na typ znakowy `char`. Możemy to zobaczyć w przykładzie:

```
#include<iostream>

using namespace std;

main(){
  int z;
    z = cin.get();  // pobranie znaku z klawiatury

  cout<<"Kod ASCII znaku: "<<z<<"\n";
  cout<<"Znak: "<<char(z);
}
```

Ciekawym przykładem łączącym odczyt i pisanie do pliku jest program, którego zadaniem jest skopiowanie pliku tekstowego. Wygląda on następująco:

```
#include<iostream>
#include<fstream>

using namespace std;

main(){

char plikA[10], plikB[10];   // ta zmienna znakowa przechowa nazwy plików

  cout<<"Podaj nazwe pliku do skopiowania: ";
  cin>> plikA;

  ifstream input(plikA);   // otwarcie strumienia do odczytu

  cout<<"Podaj nazwe kopii pliku: ";
  cin>> plikB;

  ofstream output(plikB); // otwarcie strumienia do zapisu

char c;

  while(input.get(c)){   // znaki poierane są do zmiennej 'c' (w pętli)
     output.put(c);      // i zapisywane do nowego pliku
  }
cout<<"Plik skopiowano";
}
```

Warto zauważyć, iż w powyższym programie, w celu wstawienia jednego znaku do strumienia użyliśmy funkcji `put(char)`.

W rozdziale nie skupialiśmy się na obsłudze błędów wynikających np. z otwarcia pliku, który nie istnieje lub zapisu do pliku, który został otwarty w trybie do odczytu. Osoby chcące pogłębić swoją wiedzę powinny sięgnąć w tym przypadku do literatury.

Ćwiczenia

1. Napisz program, który oblicza ilość znaków w wybranym pliku tekstowym. Pamiętaj, że w pliku znajdują się również znaki niewidoczne, np. znaki końca wiersza.
2. Napisz program, który obliczy ilość liter 'a' w wybranym pliku tekstowym.
3. Napisz program, który skopiuje plik tekstowy zastępując wszystkie znaki 'o' prze '_'.
4. Napisz program, który podwoi (skopiuje od siebie) zawartość pliku tekstowego.

Rozdział 8

Zadania powtórkowe

1. Napisz funkcję wyświetlającą na ekran pierwiastki równania kwadratowego $ax^2 2 + bx + c = 0$, gdzie a, b, c są liczbami rzeczywistymi. Delta w równaniu powinna być liczona odrębną funkcją. Przetestuj program dla przykładowych wartości a, b, c zadanych w programie lub wprowadzanych z klawiatury.
2. Napisz funkcje, które będą zwracały wartość minimalną, maksymalną oraz średnią z elementów tablicy jednowymiarowej o dowolnym rozmiarze. Przetestuj funkcje na przykładowej tablicy zawierającej liczby rzeczywiste. Napisz również funkcję wyświetlającą tablicę jednowymiarową o dowolnym rozmiarze. Wykorzystaj ją do wyświetlenia twojej przykładowej tablicy.
3. Napisz funkcję, która czytelnie wyświetli tabliczkę mnożenia w zakresie zadanym przez użytkownika. Przetestuj program dla zadanych wartości.
4. Napisz funkcję, która w czytelny sposób wyświetla macierz, która powstaje w wyniku pomnożenia dwóch macierzy przez siebie (o wymiarze 4 x 4). Przetestuj program dla przykładowej macierzy.
5. Napisz funkcję, która zwraca ilość cyfr w danej liczbie całkowitej. Przetestuj program dla przykładowych liczb wprowadzanych z klawiatury.
6. Napisz funkcję, która wyświetla wszystkie dzielniki danej liczby całkowitej. Przetestuj program dla przykładowych liczb wprowadzanych z klawiatury.
7. Napisz funkcję, która zwraca kąt pomiędzy dwoma wektorami w $\mathbb{R}^3$, wprowadzonymi do programu w formie dwóch jednowymiarowych tablic o rozmiarze 3. Przetestuj program dla przykładowych wektorów. Do obliczenia normy wektorów użyj dodatkowej funkcji.
8. Napisz funkcję, która wyświetli macierz odwrotną do zadanej macierzy 2x2. Macierz niech będzie zadana w formie tablicy dwuwymiarowej. Do obliczenia wyznacznika napisz funkcję zwracającą jego wartość.
9. Napisz funkcję wyświetlającą macierz transponowaną do zadanej macierzy 5x5. Macierz powinna być zadana w formie tablicy dwuwymiarowej o rozmiarze 5x5. Napisz również funkcję wyświetlającą tablicę dwuwymiarową o takim rozmiarze. Wykorzystaj ją do wyświetlenia twojej przykładowej tablicy.
10. Napisz funkcję sinDEG, która zwraca wartość sinusa kąta podanego w stopniach (a nie jak typowa funkcja - w radianach). W tym celu napisz również funkcję, która przelicza radiany na stopnie. Sprawdź działanie funkcji sinDEG dla zadanych wartości oraz wyświetl zestawienie, które w pierwszej kolumnie będzie zawierało stopnie od 0 do 90 (co 1), w drugiej - odpowiadające im wartości w radianach.
11. Napisz funkcję, która wyświetla rozwiązanie układu dwóch równań z dwiema niewiadomymi, zadanego w postaci macierzowej. Do obliczenia wyznaczników napisz funkcję zwracającą jego wartość.
12. Napisz funkcję, która zwraca wartość 1, gdy macierz 3x3 jest ortogonalna i 0, gdy nie posiada tej własności. Przetestuj program dla przykładowej macierzy zadanej w formie tablicy dwuwymiarowej o rozmiarze 3x3.

Rozdział 9

Rozwiązania wybranych zadań

W tym rozdziale są zaprezentowane przykładowe rozwiązania (kody programów) wybranych ćwiczeń zamieszczonych na zakończene każdego z rozdziałów. Komentarze zostały ograniczone do minimum tak, aby pobudzić czytelnika do twórzeczego analizowania kodów programów. Robiąc to warto: wskazać typy danych, zidentyfikować funkcje, znaleźć elementarne instrukcje i przeanalizować jakie zadanie realizują. Warto prześledzić program linijka po linijce odtwarzając zachowanie się komputera po każdym z tych kroków. W następnej kolejności można przepisać program, co wpłynie na naukę składni i uruchomić. Później warto modyfikować poszczególne jego fragmenty obserwując jednocześnie, jak zmienia się jego działanie.

■ Ćwiczenie 1, strona 6

```
#include<iostream>

using namespace std;

main(){

   cout << "SHORT INT " << sizeof(short int)<< "\n";
   cout << "INT " << sizeof(int)<< "\n";
   cout << "LONG INT " << sizeof(long int)<< "\n";
   cout << "CHAR " << sizeof(char)<< "\n";
   cout << "FLOAT " << sizeof(float)<< "\n";
   cout << "DOUBLE " << sizeof(double)<< "\n";
   cout << "LONG DOUBLE " << sizeof(long double)<< "\n";

}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 2, strona 6

```
#include<iostream>

using namespace std;

main(){

float liczbaRzeczywista, czescUlamkowa;
int czescCalkowita;


   cout << "Podaj liczbe rzeczywista: ";
   cin >> liczbaRzeczywista;

   czescCalkowita = (int)liczbaRzeczywista;
   czescUlamkowa = liczbaRzeczywista - (float)czescCalkowita;

   cout << "\n Czesc calkowita: "<< czescCalkowita;
   cout << "\n Czesc ulamkowa:  "<< czescUlamkowa;

}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 1, strona 8

```
#include<iostream>

using namespace std;

main(){

float r, pole_kola, pole_sfery, objetosc_kuli;
const float pi = 3.14159;

   cout << "Podaj promien: ";
   cin >> r;

   pole_kola = pi * r * r;
   pole_sfery = 4.0 * pi * r * r;
   objetosc_kuli = 4.0/3.0 * pi * r * r * r;

   cout << "\n Pole kola: \t\t"<< pole_kola;
   cout << "\n Pole sfery: \t\t"<< pole_sfery;
   cout << "\n Objetosc kuli: \t"<< objetosc_kuli;


}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 1, strona 10

```
#include<iostream>

using namespace std;

main(){

float l1,l2;

  cout<<"Podaj liczbe nr 1: ";
  cin>>l1;

  cout<<"Podaj liczbe nr 2: ";
  cin>>l2;

  if (l1>=l2) cout<<l2<<"\t"<<l1;
  else cout<<l1<<"\t"<<l2;

}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 2, strona 10

```
#include<iostream>

using namespace std;

main(){

float l;

  cout<<"Podaj liczbe: ";
  cin>>l;

  if (l>=1 && l<=6) cout<<"Wprowadzona liczba jest z przedzialu [1,6]";
  else cout<<"Wprowadzona liczba nie nalezy do przedzialu [1,6]";

}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 3, strona 10

```
#include<iostream>

using namespace std;

main(){

float l1,l2;

  cout<<"Podaj liczbe l1: ";
  cin>>l1;

  cout<<"Podaj liczbe l2: ";
  cin>>l2;

  if (l2!=0) cout<<"l1/l2= "<<l1/l2;
  else cout<<"Liczba 'l2' musi byc rozna od 0";
}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 1, strona 11

```
#include<iostream>

using namespace std;

main(){

    while (1==1) {
        cout << "*" << "\n";
    }

}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 2, strona 11

```
#include<iostream>

using namespace std;

main(){

int zgadnij = 34;   // liczba do zgadniecia
int proba;
int licznik=0;

    do{
            cout << "\nPodaj liczbe (0-100): ";
            cin >> proba;
            licznik++;

            if (proba==zgadnij) cout << "Brawo! Zgadles liczbe w "<<licznik<<" krokach.";
        else{
         if (proba<zgadnij) cout << "Za mala";
         else cout << "Za duza";
        }
    }while(proba!=zgadnij);

}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 1, strona 14

```
#include<iostream>

using namespace std;

main(){

 for (int i=1; i<=11; i=i+2){
   cout<<i<<"\n";
 }

}


\noindent
{\large $\blacksquare$ {\bf Ćwiczenie 1, strona 14 (wariant 2)}}

#include<iostream>

using namespace std;

main(){

 for (int i=1; i<=11; i++){
   if(i%2==0)  cout<<i<<"\n";
 }

}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 4, strona 14

```
#include<iostream>

using namespace std;

main(){

int i,suma=0;

    for (i=0; i<=100; i++){
        suma += i*i;
    }
  cout<<suma;
}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 4, strona 14

```
#include<iostream>

using namespace std;

main(){

int i;

    for (i=1; i<=10; i++){
      for (j=1; j<=10; j++){
         cout<<i*j<<" ";
      }
    }
}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 2 (druga seria), strona 14

```
#include<iostream>

using namespace std;

main(){

int i,suma=0;

    for (i=0; i<=100; i++){
        if (i>23 && i<46) continue;
      suma += i*i;
    }
  cout<<suma;
}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 1, strona 18

```
#include<iostream>

using namespace std;

main(){

int i=0;
int kwadraty[101];

    for(i=0; i<=100; i++){
        kwadraty[i]=i*i;
    }

    cout<<"Indeks \t Zawartosc komorki \n";

    for(i=0; i<=100; i++){
        cout<<i<<"\t"<<kwadraty[i]<<"\n";
    }
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 2, strona 18

```
#include<iostream>

using namespace std;

main(){

int tab[20]={1,7,3,8,2,9,3,5,1,7,0,4,4,1,5,3,3,7,3,1};
int sum=0, max=0;

    for(i=0; i<20; i++){
        sum += tab[i];
    }

    cout<<"Suma elementow tablicy = "<<sum<<"\n";

    for(i=0; i<20; i++){
        if(tab[i]>max) max=tab[i];
    }

    cout<<"Maksymalny element tablicy ="<<max;
}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 5, strona 18

```
#include<iostream>

using namespace std;

main(){

int wektor1[5] = {2,7,4,5,1};
int wektor2[5] = {3,7,2,7,4};

int iloczynSkalarny=0;

    for(i=0; i<5; i++){
        iloczynSkalarny += wektor1[i]*wektor2[i];
    }

    cout<<"Iloczyna skalarny wynosi "<<iloczynSkalarny;
}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 5, strona 20

```
#include<iostream>

using namespace std;

main(){

int wektor1[5] = {2,7,4,5,1};
int wektor2[5] = {3,7,2,7,4};

int iloczynSkalarny=0;

int *wskaznikWektor1;
int *wskaznikWektor2;

    wskaznikWektor1 = &wektor1[0];
    wskaznikWektor2 = &wektor2[0];

    for(i=0; i<5; i++){
        iloczynSkalarny += *(wskaznikWektor1+i) * *(wskaznikWektor2+i);
    }

cout<<"Iloczyna skalarny wynosi "<<iloczynSkalarny;
}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 2, strona 25

```
#include<iostream>

using namespace std;

int signum(float x){
    if (x<0) return -1;
        else if (x==0) return 0;
        else return 1;
}

main(){

float n;

    cout<<"Podaj liczbe: ";
    cin>>n;

    cout<<"\nsignum("<<n<<") = "<<signum(n);
}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 3, strona 25

```
#include<iostream>

using namespace std;

int potega(int m, int n){
    int temp=1;
    int i;

    for(i=1; i<=n; i++){
        temp *= m
    }

    return temp;
}

main(){

float n;

    cout<<"Podaj podstawe a: ";
    cin>>a;

    cout<<"\nPodaj wykladnik b: ";
    cin>>b;

    cout<<"\na^b = "<<potega(a,b);
}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 3, strona 25

```
#include<iostream>

using namespace std;

long int fibonacci(int n){
    if (n==0) return 0;
        else if (n==1) return 1;
         else return fibonacci(n-1) + fibonacci(n-2);
}


main(){

float n;

    cout<<"Podaj N: ";
    cin>>n;

    cout<<"\nN-ta liczba Fibonacciego to "<<fibonacci(n);
}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 8, strona 26

```
#include<iostream>

using namespace std;

float wyznacznik(float A[2][2]){
    return A[0][0]*A[1][1] - A[1][0]*A[0][1];
}

main(){

float M[2][2]={{1.2,4.3},{2.1,0.5}};


    cout<<"det(M) = "<<wyznacznik(M);
}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 2, strona 30

```
// deklaracje w ciele klasy

void zespolona::pomnoz(zespolona q);
void zespolona::podziel(zespolona q);

// definicje funkcji


void zespolona::pomnoz(zespolona q){
    double k1=x;
    double k2=y;

  x = k1 * q.x - k2 * q.y;
    y = k2 * q.x + k1 * q.y;
}

void zespolona::podziel(zespolona q){
  double k1=x;
    double k2=y;

    x = (k1 * q.x + k2 * q.y)/(q.x*q.x+q.y*q.y);
    y = (k2 * q.x - k1 * q.y)/(q.x*q.x+q.y*q.y);
}
```

Uwagi, komentarze, notatki:

■ Ćwiczenie 3, strona 36

```
#include<iostream>
#include<fstream.h>

using namespace std;

main(){

char plikA[10];
char plikB[10];

  cout<<"Podaj nazwe pliku do skopiowania: ";
  cin>> plikA;

  ifstream input(plikA);

  cout<<"Podaj nazwe pliku wyjsciowego: ";
  cin>> plikB;

  ofstream output(plikB);

char c;

  while(input.get(c)){
    if (c=='o') output.put('_');
    else output.put(c);
  }

cout<<"Plik skopiowano";
}
```

Uwagi, komentarze, notatki:

Bibliografia

[1] D. Yang, *C++ and Object Oriented Numeric Computation for Scientists and Engineers*, Springer-Verlag, New York, 2001.

[2] J. Grębosz, *Symfonia C++*, Oficyna Kallimach, Kraków, 1994.

[3] B. W. Kernighan, D. M. Ritchie, *Język ANSI C*, WN-T, Warszawa, 2003.