Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Laboratorium informatyki

Ćwiczenie nr 3. Operacje na zmiennych i instrukcje sterujące

Zagadnienia do opracowania:

- przeliczanie systemów liczbowych (dziesiętny, binarny, ósemkowy, szesnastkowy)
- zmienne lokalne i globalne
- specyfikatory *static* i *extern*
- rzutowanie i promocja typów
- instrukcja warunkowa *if-else*
- pętle (*for, while, do-while*)
- funkcja *rand()*

Spis treści

1. Cel ćwiczenia

Celem ćwiczenia jest zapoznanie z podstawowymi instrukcjami sterującymi w językach *C* oraz *C++*. Program laboratorium obejmuje zagadnienia przeliczania systemów liczbowych oraz operacji matematycznych i logicznych na podstawowych typach zmiennych w językach *C* i *C++*.

2. Wprowadzenie

2.1. Systemy liczbowe

Powszechnie stosowaną metodą zapisu liczb całkowitych w systemach komputerowych jest system pozycyjny. Określa on, że cyfra znajdująca się na danej pozycji w ciągu (tworzącym liczbę) stanowi wielokrotność potęgi liczby zwanej ***bazą systemu*** (np. 2 w systemie binarnym, 10 w systemie dziesiętnym, 16 w systemie heksadecymalnym, itp.). Niech $\boldsymbol{C_i}$ oznacza cyfrę na *i-tej* pozycji w liczbie, liczonej od prawej strony. Wartość (dziesiętna) liczby *n-cyfrowej* w systemie pozycyjnym o bazie $\boldsymbol{b}$ opisana jest wyrażeniem (1). Stąd, aby przeliczyć liczbę zapisaną w systemie pozycyjnym o podstawie $\boldsymbol{b}$ na system dziesiętny należy zsumować wszystkie iloczyny $C_i \cdot b^{i-1}$.

$$\sum_{i=1}^{n} C_i \cdot b^{i-1} \tag{1}$$

Przykłady:

$$12345_{(10)} = 1 \cdot 10^4 + 2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0$$

$$10110100_{(2)} = 1 \cdot 2^7 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^2 = 128 + 32 + 16 + 4 = 180_{(10)}$$

Aby przeliczyć liczbę zapisaną w systemie dziesiętnym na inny system pozycyjny o podstawie $\boldsymbol{b}$ można posłużyć się następującym algorytmem:

1. podziel liczbę przez podstawę $\boldsymbol{b}$
2. w i-tym kroku zapisz resztę z dzielenia na i-tej pozycji w liczbie
3. jeśli wynik dzielenia jest różny od 0, to wróć do punktu 1.

Przykład (przeliczanie $26_{(10)}$ na system binarny):

$$26|2 = 13r0$$
$$13|2 = 6r1$$
$$6|2 = 3r0$$
$$3|2 = 1r1$$
$$1|2 = 0r1$$

$$26_{(10)} = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 11010_{(2)}$$

Zarówno język *C*, jak i *C++* umożliwiają przypisanie do zmiennych wartości liczbowych o różnych bazach (listing 1).

```
1 int a = 180; // system dziesietny
2 int b = 0b10110100; // system binarny (prefiks 0b)
3 int c = 0264; // system osemkowy (prefiks 0)
4 int d = 0xB4; // system szesnastkowy (prefiks 0x)
```

Listing 1. Inicjalizacja zmiennej z wykorzystaniem różnych systemów liczbowych

System binarny znalazł powszechne zastosowanie w elektronice cyfrowej, m.in. w opisie układów bramek logicznych. System ósemkowy jest używany np. do nadawania uprawnień dostępu do plików i katalogów w systemach operacyjnych z rodziny Unix (polecenie *chmod*). System szesnastkowy służy do zapisu kodu maszynowego, adresów sprzętowych MAC, czy kodów barw stosowanych np. w HTML czy CSS.

2.2. Rodzaje zmiennych

Zarówno w języku *C*, jak i *C++* rozróżniamy zmienne ***lokalne*** oraz ***globalne***. Różnią się one zasięgiem (widoczność, zakres ważności), rodzajem wiązania, czasem życia, występowaniem mechanizmu domyślnej inicjalizacji oraz segmentem pamięci, w którym są alokowane. Rodzaje zmiennych porównano w tabeli 1. Dodatkowo uwzględniono w niej użycie specyfikatora ***static***.

Tabela 1. Porównanie rodzajów zmiennych w językach *C/C++*

Rodzaj zmiennej	**Zasięg (widoczność)**	**Rodzaj wiązania**	**Czas życia**	**Domyślna inicjalizacja**	**Segment pamięci**
lokalna	w obrębie bloku	brak	w obrębie bloku	brak	stos
globalna	cały program	zewnętrzne	cały program	zerowanie	.data[1]
statyczna lokalna	w obrębie bloku	brak	cały program[2]	zerowanie	.data[1]
statyczna globalna	w obrębie jednostki translacji	wewnętrzne	cały program	zerowanie	.data[1]

[1]Dotyczy zainicjalizowanych zmiennych

[2]Od momentu inicjalizacji do końca działania programu

Zmienne lokalne są widoczne (można się do nich odwołać) w obrębie bloku [3], w którym zostały zdeklarowane, od miejsca (linii kodu) ich deklaracji. W momencie zdefiniowania zmiennej lokalnej zostaje zaalokowana potrzebna pamięć w segmencie zwanym ***stosem***. Pamięć ta jest automatycznie zwalniana, gdy program opuści blok, w którym zdefiniowana została zmienna lokalna. Z tego powodu zmienne lokalne zwane są również ***zmiennymi automatycznymi***. Zmienne lokalne nie są domyślnie inicjalizowane. Oznacza to, że jeżeli nie przypiszemy do zmiennej lokalnej konkretnej wartości, to jej wartość będzie **niezdefiniowana** (będzie równa wartości aktualnie przechowywanej na stosie pod adresem zmiennej).

Zmienne globalne są widoczne w obrębie całej aplikacji. Deklarowane są poza ciałami funkcji. Zgodnie ze standardem *ANSI C* zmienne globalne są domyślnie zerowane (jeżeli nie zainicjalizujemy ich inną wartością), jeszcze przed wywołaniem funkcji *main()*. Stąd też czas życia zmiennych globalnych jest równy czasowi wykonywania programu **i mogą zostać zainicjalizowane wyłącznie za pomocą wyrażeń stałych**. Jeszcze niezainicjalizowane zmienne globalne przechowywane są w segmencie pamięci ***.bss***, a po inicjalizacji w segmencie ***.data***.

Lokalne zmienne statyczne to zmienne lokalne zdeklarowane (wewnątrz bloku) przy użyciu słowa kluczowego ***static***. Nie wpływa to na ich widoczność (a więc również na rodzaj wiązania), ale zmienia czas życia (od momentu inicjalizacji do końca działania aplikacji). Inicjalizacja zachodzi jednorazowo, przy pierwszym wywołaniu zmiennej (domyślnie wartością 0). Jest to tak zwana ***leniwa inicjalizacja zmiennej*** i stosowana jest często do inicjalizacji lokalnych zmiennych o znacznym rozmiarze, których każdorazowe tworzenie i zwalnianie byłoby kosztowne. W związku z powyższym, ***lokalne zmienne statyczne*** utrzymują swoją wartość między wywołaniami funkcji. Innymi słowy, zmienna utrzymuje wartość zapisaną po poprzednim

[3] funkcja lub jej fragment umieszczony w obrębie dodatkowych nawiasów klamrowych

wykonaniu funkcji, w której została zdeklarowana. Zmienne zdeklarowane przy użyciu słowa kluczowego ***static*** są alokowane w tych samych segmentach pamięci co zmienne globalne.

Globalne zmienne statyczne to zmienne globalne zdeklarowane przy użyciu słowa kluczowego ***static***. Od ***zmiennych globalnych*** różnią się rodzajem wiązania (***wewnętrzne***) – można odwołać się do nich jedynie w jednostce kompilacji, w której zostały zdeklarowane.

W języku *C* ***stałe*** to pewien rodzaj zmiennych, zdeklarowanych przy użyciu słowa kluczowego ***const***. Kompilator pilnuje, aby nie doszło do jawnej modyfikacji wartości stałej w kodzie programu (ale może być to osiągnięte przez modyfikację fragmentu pamięci, w której przechowywana jest stała). Z tego względu nie jest możliwe rozdzielenie deklaracji i definicji stałej. W języku *C++* wyrażenia opatrzone słowem kluczowym ***const*** to stałe w pełnym tego słowa znaczeniu. Alokowane są w segmencie pamięci tylko do odczytu (*read-only*) ***.rodata***, a ich wykorzystanie w kodzie może być optymalizowane przez kompilator. **Z tego względu zaleca się, aby wszystkie wyrażenia, których wartość nie będzie modyfikowana w trakcie działania programu, były deklarowane przy użyciu kwalifikatora *const*.**

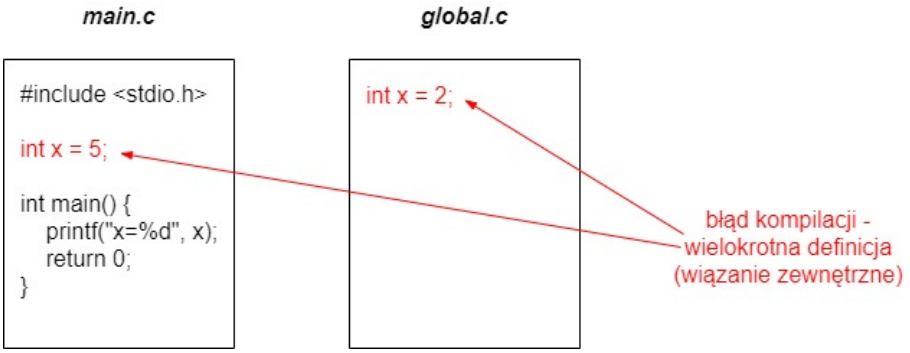
Przykłady różnych zmiennych przedstawiono na listingu 2. Symbol ***\n***, wykorzystany w funkcji *printf()*, określa znak przejścia do nowej linii.

```
1 #include <stdio.h>
2
3 // Zmienna globalna
4 int globalVar = -3;
5 // Niezainicjalizowana zmienna globalna
6 unsigned long uninitializedGlobalVar;
7 // Statyczna zmienna globalna
8 static float staticGlobalVar = 1.5;
```

```
9
10 // Globalna stala
11 const short globalConst = 4;
12
13 int main() {
14   // Statyczna zmienna lokalna
15   static double staticLocalVar = 2.0;
16   // Zmienna lokalna
17   long localVar = 125;
18   // Niezainicjalizowana zmienna lokalna
19   int uninitializedLocalVar;
20
21   // Lokalna stala
22   const unsigned int localConst = 11;
23
24   printf("Global variable: %d\n", globalVar);
25   printf("Uninitialized global variable: %d\n",
      uninitializedGlobalVar);
26   printf("Static global variable: %f\n",
      staticGlobalVar);
27   printf("Global constant: %d\n", globalConst);
28   printf("Static local variable: %f\n",
      staticLocalVar);
29   printf("Local variable: %d\n", localVar);
30   // Wyswietli losowa wartosc
31   printf("Uninitialized local variable: %d\n",
      uninitializedLocalVar);
32   printf("Local constant: %u\n", localConst);
33   return 0;
34 }
```

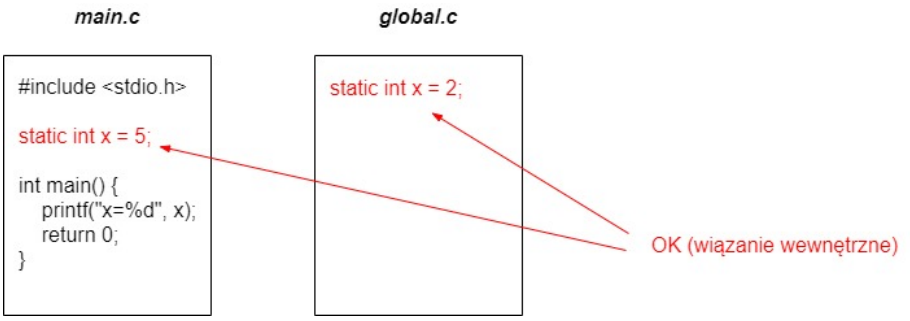
Listing 2. Różne rodzaje zmiennych w języku *C/C++*

Z pojęciem ***zasięgu (widoczności, zakresu ważności)*** łączy się bezpośrednio pojęcie ***wiązania*** (*ang. linkage*). **Wiązanie zewnętrzne** określa, że do danej zmiennej można odwołać się z dowolnej jednostki translacji. **Wiązanie wewnętrzne** oznacza, że dana zmienna jest widoczna wyłącznie w jednostce translacji, w której została zdeklarowana. **Brak wiązania** jest równoważny z ograniczeniem zasięgu zmiennej do bloku, w którym została zdeklarowana. Ponieważ zmienne globalne charakteryzują się ***wiązaniem zewnętrznym***, utworzenie niezależnych zmiennych globalnych o tej samej nazwie w różnych jednostkach translacji (plikach źródłowych) poskutkuje zgłoszeniem przez konsolidator błędu wielokrotnej definicji (*multiple definition of (...) first defined here*) (rys. 2.1).



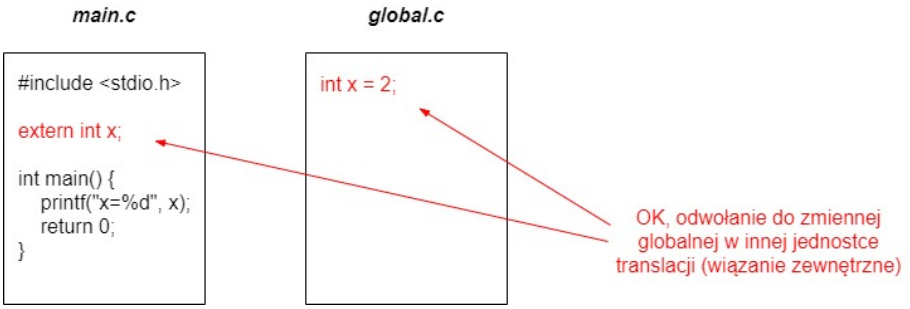
Rys. 2.1. Błąd wielokrotnej definicji (wiązanie zewnętrzne)

Można to rozwiązać przez zmianę rodzaju wiązania – wykorzystując słowo kluczowe ***static*** (rys. 2.2).



Rys. 2.2. Zastosowanie specyfikatora *static* (wiązanie wewnętrzne)

Jeżeli spróbujemy odwołać się do zmiennej globalnej zdeklarowanej w innej jednostce translacji, kompilator zgłosi błąd, że zmienna do której się odwołujemy nie została zdeklarowana. Jest to oczekiwanie zachowanie, jeżeli stosowana jest kompilacja rozłączna. Aby to rozwiązać należy posłużyć się słowem kluczowym ***extern*** (rys. 2.3).



Rys. 2.3. Zastosowanie specyfikatora *extern* (wiązanie zewnętrzne)

W językach *C* i *C++* możliwe jest deklarowanie zmiennych globalnych i lokalnych o tej samej nazwie w obrębie jednej jednostki translacji. Wówczas, odwołując się do identyfikatora zmiennej odwołujemy się do zmiennej **lokalnej**. Aby odwołać się do zmiennej globalnej należy posłużyć się słowem kluczowym ***extern*** w języku *C* lub prościej, ***operatorem ::*** w języku *C++*. Mechanizm ten nosi nazwę ***przesłaniania zmiennych*** (*ang. variable shadowing*) i został przedstawiony na listingach 3 oraz 4.

```c
#include <stdio.h>

// Zmienna globalna
double number = 2.0;

int main() {
    // Zmienna lokalna
    double number = 5.5;

    // Przeslanianie zmiennych
    printf("Local variable: %f\n", number);

    {
        extern double number;
        printf("Global variable: %f\n", number);
    }
    return 0;
}
```

Listing 3. Przesłanianie zmiennych w języku *C*

```c
#include <stdio.h>

// Zmienna globalna
double number = 2.0;

int main() {
    // Zmienna lokalna
    double number = 5.5;

    // Przeslanianie zmiennych
    printf("Local variable: %f\n", number);
```

```
12      printf("Global variable: %f\n", ::number);
13      return 0;
14  }
```

Listing 4. Przesłanianie zmiennych w języku *C++*

Zmienne globalne powinny być stosowane z rozwagą. Często, zastosowanie zmiennej globalnej zamiast lokalnej może przysporzyć więcej szkody niż pożytku, dlatego też zaleca się **ograniczanie wykorzystania zmiennych globalnych w kodzie aplikacji**. Jeżeli jest to możliwe, należy zapewnić **wiązanie wewnętrzne** za pomocą słowa kluczowego ***static***.
Ograniczenia i zastosowanie zmiennych globalnych:

+ stosowane zamiast zmiennych lokalnych w mikrokontrolerach o małej pamięci stosu (wykorzystanie znacznej liczby zmiennych lokalnych mogłoby doprowadzić do jego ***przepełnienia*** (*ang. stack overflow*))
+ stosowane do przekazywania zmiennych między funkcjami w architekturach procesorów niewspierających dynamicznej alokacji pamięci
– mogą być modyfikowane przez każdą funkcję programu, co może doprowadzić do konfliktu modyfikacji i w rezultacie do błędnego działania aplikacji
– w przypadku aplikacji wielowątkowych, mogą być modyfikowane przez dowolny wątek, co prowadzi do problemu synchronizacji dostępu do zmiennej
– są przesłaniane przez zmienne lokalne, przez co łatwo jest nieopatrznie odwołać się do niewłaściwej zmiennej

2.3. Operacje na zmiennych

Wykonując operacje na zmiennych należy mieć na uwadze, że tak samo jak w matematyce istotna jest kolejność wykonywanych działań. W tabeli 2 uszeregowano listę operatorów dostępnych w językach *C* oraz *C++* według ich priorytetu (*ang. operator precedence*). Dodatkowo zamieszczono informację o ich łączności (*ang. associativity*), tj. który z dwóch operatorów o tym samym priorytecie zostanie wykonany jako pierwszy: stojący z lewej strony zmiennej (łączność lewostronna) lub z prawej strony (łączność prawostronna). Jeżeli nie pamiętamy, który z dwóch operatorów ma większy priorytet, możemy posłużyć się dodatkowymi nawiasami okrągłymi *()*. Przez *x*, *y*, *z* oznaczono przykładowe identyfikatory zmiennych, słowem *type* zastąpiono dowolny typ zmiennych w językach *C/C++*.

Tabela 2. Priorytety i łączność operatorów w językach *C/C++* [1][2]

Priorytet	**Operator**	**Opis**	**Łączność**
1	:: [*C++*]	rozdzielczość zakresu	lewostronna
2	x++	postinkrementacja	
	x−−	postdekrementacja	
	type(x)	rzutowanie typu	
	type{x} [od *C++11*]		
	$x()$	wywołanie funkcji	
	$x[]$	indeks	
	.	dostęp do składowych	
	−>	klasy (struktury)	
3	++x	preinkrementacja	prawostronna
	−−x	predekrementacja	
	+x	promocja liczby	
	−x	negacja liczby	
	!	logiczna negacja (NOT)	
	∼	bitowa negacja (NOT)	

	(*type*)x	rzutowanie typu (w stylu C)	
	$*x$	dereferencja (wyłuskanie)	
	$\&x$	adres	
	sizeof	rozmiar (w bajtach)	
	new [$C++$] new[] [$C++$]	dynamiczna alokacja pamięci	
	delete [$C++$] delete[] [$C++$]	dynamiczne zwolnienie pamięci	
4	.* $->*$	dostęp do składowych klasy (struktury) przez wskaźnik	
5	x * y x / y x % y	mnożenie dzielenie dzielenie z resztą	
6	x + y x − y	dodawanie odejmowanie	
7	<< >>	przesunięcie bitowe w lewo przesunięcie bitowe w prawo	
8	<=> [od $C++20$]	trójwynikowe porównanie	
9	< <= > >=	mocna relacja mniejszości słaba relacja mniejszości silna relacja większości słaba relacja większości	lewostronna
10	== !=	jest równe jest różne	
11	&	iloczyn bitowy (AND)	
12	^	bitowa alternatywa wykluczająca (XOR)	
13	|	suma bitowa (OR)	
14	&&	iloczyn logiczny (AND)	
15	||	suma logiczna (OR)	
	$x\ ?\ y : z$	warunek trójskładnikowy	

	throw [C++]	rzucenie wyjątku	
	=	przypisanie	
	+=	przypisanie i dodawanie	
	-=	przypisanie i odejmowanie	
	*=	przypisanie i mnożenie	
	/=	przypisanie i dzielenie	
	%=	przypisanie i dzielenie z resztą	
	<<=	przypisanie i przesunięcie bitowe w lewo	
	>>=	przypisanie i przesunięcie bitowe w prawo	
	&=	przypisanie i bitowa koniunkcja	
	^=	przypisanie i bitowa alternatywa wykluczająca	
	|=	przypisanie i bitowa alternatywa	
17	,	przecinek	lewostronna

Przykład 1.

```
int x = 3;
int y = ++x * 2;
printf("y: %d", y);
```

Rezultatem operacji $++x*2$ będzie wynik **8**, ponieważ operator inkrementacji (zwiększenia wartości o 1) ma większy priorytet (3) niż operator mnożenia (5). Operację tę można rozpisać w równoważny sposób:

```
int x = 3;
int y = (x + 1) * 2;
printf("y: %d", y);
```

Przykład 2.

```
int x = 2;
int y = 2 * x % 3;
printf("y: %d", y);
```

Rezultatem operacji $2 * x \ \% \ 3$ będzie wynik **1**. Operatory mnożenia i dzielenia z resztą mają ten sam priorytet (5). Ponieważ operatory te mają *łączność lewostronną*, to kod ten może być równoważnie zapisany w następujący sposób:

```
int x = 2;
int y = (2 * x) % 3;
printf("y: %d", y);
```

2.4. Rzutowanie i promocja typów

Wykonując operacje na różnych typach danych (np. dodając liczby całkowite i zmiennoprzecinkowe) obserwujemy działanie mechanizmu ***promocji typów***. Polega on na uwspólnieniu typów argumentów operatora matematycznego. Promocja wykonywana jest według następującego algorytmu:

1. jeżeli jeden z argumentów jest typu ***double***, to drugi argument przekształcany jest do typu ***double***, w przeciwnym wypadku
2. jeżeli jeden z argumentów jest typu ***float***, to drugi argument przekształcany jest do typu ***float***, w przeciwnym wypadku
3. jeżeli jeden z argumentów jest typu ***long***, to drugi argument przekształcany jest do typu ***long***, w przeciwnym wypadku
4. jeżeli jeden z argumentów jest typu ***int***, to drugi argument przekształcany jest do typu ***int***.

Ponadto, na zmiennych typu ***char*** czy ***short*** przeprowadzana jest ***automatyczna promocja do typu int***, kiedy stają się argumentami operatora matematycznego (np. dodawania czy mnożenia). Jeżeli typ ***int*** może reprezentować wszystkie wartości promowanego typu (obejmuje wartość maksymalną), to zmienna konwertowana jest na typ ***int***, w przeciwnym razie na typ ***unsigned int***.

Promocja typów jest przykładem ***niejawnego rzutowania (konwersji)*** typów. Niejawnego, tzn. dokonywanego przez kompilator w sposób automatyczny. Przykład obrazujący ten mechanizm przedstawiono na listingu 5. Wynikiem uruchomienia przedstawionej aplikacji będzie wyświetlenie *z: 0.000000*. Doszło tutaj do promocji typu wyniku operacji dzielenia. Ponieważ oba argumenty operatora dzielenia są liczbami typu ***int***, to kompilator automatycznie założył, że wynik z dzielenia również ma być liczbą całkowitą (obcinając część ułamkową 0.333333).

```c
#include <stdio.h>

int main() {
    int x = 1;
    int y = 3;
    float z = x / y;
    printf("z: %f", z);
    return 0;
}
```

Listing 5. Niejawne rzutowanie typów

Aby otrzymać oczekiwany wynik dzielenia (liczbę zmiennoprzecinkową) wystarczy zmienić przynajmniej jeden z typów zmiennych x lub y na typ ***float***. Wówczas, zgodnie z algorytmem promocji typów, druga zmienna zostanie niejawnie przekonwertowana na typ ***float***, a wynik operatora dzielenia

będzie liczbą zmiennoprzecinkową. Można również, na potrzeby operacji dzielenia, przeprowadzić ***jawne rzutowanie*** typu jednego z argumentów na typ ***float***. Wówczas, domyślny typ zmiennej w kolejnych operacjach pozostanie niezmieniony. ***Jawne rzutowanie*** przedstawiono na listingu 6. Na podstawie tabeli 2 można wywnioskować, że priorytet operatora ***(float)*** (3) jest większy niż operatora dzielenia (5), dlatego zmienna x zostanie w pierwszej kolejności przekonwertowana na typ ***float***, a następnie podzielona przez liczbę y. Ponieważ jeden z argumentów operatora dzielenia (x) jest teraz typu zmiennoprzecinkowego, to nastąpi automatyczna promocja zmiennej y do typu ***float***.

```
#include <stdio.h>

int main() {
    int x = 1;
    int y = 3;
    float z = (float) x / y;
    printf("z: %f", z);
    return 0;
}
```

Listing 6. Jawne rzutowanie typów

Zapis ***(float) x*** można stosować wymiennie z zapisem ***float(x)***. Te dwa operatory różnią się jedynie priorytetem wykonania (patrz tabela 2). Pierwszy zapis to tak zwane „rzutowanie w stylu *C*". W języku *C++* do jawnego rzutowania typu zastosowano by operator ***static_cast<float>(x)***. Operator rzutowania w języku *C++* ma kilka zalet w stosunku do operatora rzutowania w stylu *C*:

- możliwość rzutowania jednego typu w drugi jest sprawdzana na etapie kompilacji (operator rzutowania w stylu *C* zgłosi błąd na etapie działania programu);

- jest lepiej widoczny w kodzie;
- łatwiej zrozumieć intencję programisty.

Podczas jawnego rzutowania typów należy mieć na uwadze, że konwersja zmiennej typu większego na mniejszy (np. ***long*** na ***short***) może się wiązać z utratą danych, ponieważ zmienna mniejszego typu nie zapewnia możliwości przechowania wartości numerycznych z tego samego zakresu (patrz: tabela 1 [Ćw. 2]).

2.5. Instrukcje sterujące

Instrukcje sterujące to zbiór mechanizmów języka, umożliwiających zmianę kolejności oraz warunkowe wykonanie określonych operacji. Pierwszym typem instrukcji sterujących są ***instrukcje warunkowe***, których reprezentantem jest instrukcja ***if–else***. Składnia tej instrukcji została przedstawiona na listingu 7.

```
1 if (condition) {
2     // Jezeli warunek condition jest prawdziwy, ten
    blok kodu zostanie wykonany
3 } else if (second_condition) {
4     // Jezeli warunek condition jest falszywy i
    jednoczesnie warunek second_condition jest
    prawdziwy, ten blok kodu zostanie wykonany
5 } else {
6     // W innym przypadku ten blok kodu zostanie
    wykonany
7 }
```

Listing 7. Składnia instrukcji warunkowej ***if–else***

Przykład wykorzystania instrukcji ***if–else*** w funkcji sprawdzającej czy przekazana liczba jest parzysta (*isEven()*) przedstawiono na listingu 8. W tym

przypadku nawiasy klamrowe po instrukcji ***if*** mogą zostać opuszczone – wykonana zostanie wyłącznie pierwsza instrukcja po instrukcji warunkowej (tu: przypisanie wartości logicznej ***true*** do zmiennej *result*).

```
bool isEven(int number) {
    // Zmienna typu logicznego przechowujaca wynik
    bool result = false;

    // Sprawdzenie reszty z dzielenia przez 2
    if (number % 2 == 0)
        result = true;
    return result;
}
```

Listing 8. Funkcja *isEven()* sprawdzająca parzystość przekazanej liczby

Kilka wyrażeń warunkowych może być łączonych za pomocą operatorów logicznych (NOT, OR, AND – patrz tabela 2):

```
if ((x > 0 && x < 10) || (x == 99)) {
    // Zrob cos
}
```

W językach *C* i *C++* **wyrażenie jest prawdziwe, jeżeli jego wartość jest różna od zera**, również jeżeli jest ujemna. Standardowo, wartość ***true*** zrzutowana na liczbę całkowitą wynosi **1**, a wartość ***false*** **0**. Aby posługiwać się typem ***bool*** w języku *C* należy załączyć plik nagłówkowy ***stdbool.h***.

Częstym błędem popełnianym przez początkujących programistów jest mylenie operatorów ***porównania*** (==) i ***przypisania*** (=):

```
if (x = 2) { // zamiast x == 2
    // Zrob cos
}
```

Zamiast sprawdzenia czy wartość zmiennej x jest równa 2 ($x == 2$), wykonujemy przypisanie wartości 2 do zmiennej x ($x = 2$). Kod wewnątrz bloku ***if*** zostanie zawsze wykonany, ponieważ do instrukcji warunkowej ***if*** przekazana zostanie wartość zmiennej x, czyli w tym przypadku liczba 2. Liczba 2 jest różna od 0 (***true***), zatem warunek wejścia do bloku zostanie spełniony.

Drugą grupą instrukcji sterujących są ***pętle***. Służą one wielokrotnemu wykonaniu pewnego ciągu operacji. W językach *C* i *C++* wyróżniamy trzy podstawowe pętle: ***for***, ***while*** oraz ***do–while***. Pętla ***while*** wykonuje instrukcję zawartą w bloku (lub pierwszą instrukcję występującą po instrukcji ***while*** w przypadku pominięcia nawiasów klamrowych) tak długo, jak warunek *condition* jest prawdziwy (listing 9). Jeżeli warunek ten na wstępie jest fałszywy, to nie zostanie wykonana ani jedna ***iteracja*** pętli.

```
while (condition) {
    // Wykonuj tak dlugo, jak warunek condition jest
    prawdziwy
}

// Wyswietl wszystkie liczby naturalne od 1 do 9
int x = 1;
while (x < 10) {
    printf("%d\n", x++);
}
```

Listing 9. Składnia pętli ***while***

Pętla ***do–while*** różni się od pętli ***while*** tym, że warunek *condition* jest sprawdzany na koniec iteracji pętli. Zatem **przynajmniej jedna iteracja zostanie wykonana** nawet, jeśli warunek od początku jest fałszywy. Składnię pętli ***do–while*** przedstawiono na listingu 10.

```
do {
    // Sprawdz warunek condition na koniec kazdej iteracji
    // Wykonuj tak dlugo, jak warunek condition jest prawdziwy
} while (condition);
```

Listing 10. Składnia pętli ***do–while***

Pętla ***for*** zawiera w swojej składni trzy wyrażenia: *initializer*, *condition*, *post_iteration*, rozdzielone średnikami (listing 11). Pierwsze wyrażenie to instrukcja, która ma wykonać się **przed pierwszą iteracją pętli**. Najczęściej jest to inicjalizacja zmiennej stanowiącej iterator (licznik iteracji) pętli. Drugie wyrażenie to **warunek zakończenia pętli** – pętla będzie wykonywać się tak długo jak warunek *condition* będzie prawdziwy. Ostatnie wyrażenie to instrukcja, która będzie wykonywana **po każdej iteracji**. Najczęściej jest to odpowiednio inkrementacja lub dekrementacja licznika.

```
for (initializer; condition; post_iteration) {
    // Wykonuj tak dlugo, jak warunek condition jest prawdziwy
}

// Wyswietl wszystkie liczby naturalne od 9 do 1
int i;
for (i = 9; i > 0; --i) {
    printf("%d\n", i);
```

```
}
```

Listing 11. Składnia pętli ***for***

Pętle w języku *C/C++* możemy ***zagnieżdżać*** (*and. nested loops*), tj. umieszczać jedną pętlę wewnątrz drugiej. Przykładową funkcję wypisującą na konsolę tabliczkę mnożenia o rozmiarze *n x n* przedstawiono na listingu 12.

```
void printMultiplicationTable(int n) {
    int i, j;

    for (i = 1; i <= n; ++i) {
        for (j = 1 ; j <= n; ++j) {
            // Wyswietl wynik mnozenia oraz
   pojedynczy znak tabulacji
            printf("%d\t", i * j);
        }
        printf("\n"); // Przejdz do nowej linii
    }
}
```

Listing 12. Przykład zagnieżdżenia pętli w językach *C/C++*

Do wcześniejszego przerwania pętli lub pominięcia określonej iteracji wykorzystuje się odpowiednio słowa kluczowe ***break*** oraz ***continue***. Przykład zastosowania instrukcji ***break*** i ***continue*** przedstawiono na listingu 13. Funkcja *listEvenNumbers()* wypisuje wszystkie liczby parzyste w zakresie od 0 do *range*. Dozwolony przedział jest ograniczony z góry przez liczbę 1000.

```
void listEvenNumbers(unsigned int range) {
    int i;
    for (i = 0; i < range; ++i) {
```

```
4           // Pomin nieparzyste iteracje
5           if (i % 2 != 0)
6               continue;
7
8           // Wypisz liczbe
9           printf("%d\n", i);
10
11          // Przerwij petle, jesli iterator przekroczy
        gorny limit
12          if (i >= 1000)
13              break;
14      }
15  }
```

Listing 13. Zastosowanie instrukcji *break* i *continue*

Czasami, w szczególności w przypadku programowania mikrokontrolerów nieposiadających systemu operacyjnego, można spotkać się z instrukcją ***for(;;)***. Jest to instrukcja, która będzie nieskończenie długo przechodzić do kolejnej iteracji, równocześnie nie wykonując żadnej dodatkowej operacji. Zastosowanie takiej instrukcji na końcu ciała funkcji *main()* służy niedopuszczeniu do wyjścia z funkcji *main()*, co byłoby równoważne z zakończeniem programu (stąd też w oprogramowaniu mikrokontrolerów można napotkać funkcję *main()*, która nic nie zwraca – *void main()*). Mikrokontroler ma wykonywać ciągle jeden program, a poszczególne peryferia obsługiwane są za pomocą ***przerwań systemowych***, które przekazują sterowanie z funkcji *main()* do określonego podprogramu obsługi przerwania.

2.6. Generator liczb pseudolosowych

Generator liczb pseudolosowych (*Pseudo-Random Number Generator, PRNG*) to program, który na podstawie informacji wejściowej (*ziarno, ang. seed*) generuje **deterministyczny** ciąg liczb, którego właściwości są podobne do wła-

ściwości ciągu liczb losowych. Aby wygenerować taki ciąg w języku *C*, można posłużyć się funkcją ***rand()***, zdeklarowaną w pliku nagłówkowym biblioteki standardowej **stdlib.h**. Funkcja ***rand()*** zwraca pseudolosową liczbę całkowitą z zakresu od *0* do *RAND_MAX* (wartość zdefiniowana w **stdlib.h**). Jeżeli chcemy ograniczyć zakres generowanych liczb, możemy posłużyć się operatorem modulo (dzielenia z resztą):

```
1 // Wygeneruj pseudolosowa liczbe calkowita z zakresu
      [0, 999]
2 int x = rand() % 1000;
3 // Wygeneruj pseudolosowa liczbe calkowita z zakresu
      [5, 14]
4 int y = 5 + rand() % 10;
```

Do inicjalizacji **PRNG** służy funkcja ***srand()*** przyjmująca *ziarno* w postaci nieujemnej liczby całkowitej. Inicjalizując generator stałą wartością otrzymamy tę samą pseudolosową liczbę przy każdym kolejnym uruchomieniu aplikacji. Możemy również zainicjalizować go wartością zmieniającą się z każdym kolejnym uruchomieniem programu, aby losować różne liczby. Najczęściej osiąga się to przez wywołanie funkcji *srand()* z wartością zwróconą przez funkcję *time()*, zdeklarowaną w pliku nagłówkowym **time.h**. Funkcja *time()* zwraca liczbę sekund, która upłynęła od północy 01.01.1970. Przykład inicjalizacji generatora liczb pseudolosowych przedstawiono na listingu 14.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int main() {
6     srand(time(NULL));
7     int x = rand();
```

```
8       printf("x: %d", x);
9       return 0;
10  }
```

Listing 14. Inicjalizacja generatora liczb pseudolosowych z wykorzystaniem funkcji *time()*

3. Program ćwiczenia

Zadanie 1. Napisz funkcję ***void zeros(float a, float b, float c)***, która oblicza, a następnie wyświetla na ekranie miejsca zerowe funkcji kwadratowej $f(x) = ax^2 + bx + c$ w dziedzinie liczb rzeczywistych, na podstawie zadanych współczynników ***a***, ***b***, ***c***. Deklaracja funkcji została umieszczona w pliku ***zeros.h***. Umieść definicję funkcji w pliku ***zeros.c***. Zaimplementowaną funkcję wywołaj w funkcji ***main()*** i zweryfikuj poprawność jej działania.
Podpowiedź: pierwiastek kwadratowy można obliczyć wykorzystując funkcję sqrt(), zdeklarowaną w pliku nagłówkowym ***math.h***

Zadanie 2. Wykorzystując dowolną pętlę, napisz funkcję ***void decimalToBinary(unsigned int number)***, która przyjmuje liczbę całkowitą (w systemie dziesiętnym) i wyświetla na konsoli jej odpowiednik w systemie binarnym. Deklaracja funkcji została umieszczona w pliku ***conversion.h***. Umieść definicję funkcji w pliku ***conversion.c***. Zaimplementowaną funkcję wywołaj w funkcji ***main()*** i zweryfikuj poprawność jej działania.
Podpowiedź: funkcja printf() nie posiada ciągu formatującego do wyświetlania reprezentacji binarnej liczb. Aby wyświetlić przekonwertowaną liczbę poprawnie za pomocą funkcji printf(), zapisz ją do zmiennej jako liczbę dziesiętną w następującej postaci (przykład dla liczby binarnej 101011):

$$101011 = 1 \cdot 10^5 + 0 \cdot 10^4 + 1 \cdot 10^3 + 0 \cdot 10^2 + 1 \cdot 10^1 + 1 \cdot 10^0$$

Zadanie 3. Algorytm ***RSA*** jest przykładem kryptograficznego algorytmu asymetrycznego wykorzystywanego m.in. w zabezpieczaniu komunikacji Internetowej. Kryptografia asymetryczna, zwana również kryptografią klucza publicznego, zakłada wykorzystanie dwóch powiązanych ze sobą kluczy. Jeden z kluczy (klucz publiczny) jest ujawniany bez zagrożenia utraty bezpieczeństwa danych. Drugi z kluczy (klucz prywatny, klucz deszyfrujący) jest niejawny i nie jest możliwe jego łatwe odtworzenie na podstawie klucza publicznego. Jego obliczenie sprowadza się do złamania operacji jednokierun-

kowej. Bezpieczeństwo algorytmu *RSA* bazuje na problemie faktoryzacji (dla danego n znaleźć takie liczby pierwsze p i q, że $n = p \cdot q$). Procedura obliczania kluczy jest następująca [3]:

1. losowo wybierz dwie duże liczby pierwsze p i q
2. oblicz $n = p \cdot q$
3. wybierz nieparzystą liczbę e względnie pierwszą z $p-1$ oraz $q-1$, tj. $gcd(e,\ p-1) = gcd(e,\ q-1) = 1$
4. oblicz sekret d z równania (2)

$$d = e^{-1}\ mod\ \phi(n), \tag{2}$$

   gdzie $\phi(n) = (p-1)(q-1)$ – tocjent Eulera, a e^{-1} oznacza ***odwrotność modularną*** liczby e
5. opublikuj parę (e, n) jako klucz publiczny, para (d, n) stanowi klucz prywatny.

Zadanie polega na wyznaczeniu liczb ***p***, ***q*** i ***e*** na potrzeby obliczenia kluczy algorytmu *RSA*. W pliku nagłówkowym **rsa_utils.h** zdeklarowano funkcje ***int gcd(unsigned int x, unsigned int y)*** – obliczającą największy wspólny dzielnik liczb x i y oraz ***bool isPrime(unsigned int x)*** – sprawdzającą czy przekazana liczba jest liczbą pierwszą. Utwórz plik źródłowy **rsa_utils.c** i zaimplementuj funkcje *gcd()* oraz *isPrime()*. W funkcji *main()* zainicjalizuj generator liczb pseudolosowych, a następnie za jego pomocą wylosuj liczby ***p***, ***q*** i ***e*** takie, że ***isPrime(p) == isPrime(q) == true*** oraz ***gcd(e, p - 1) == gcd(e, q - 1) == 1***.

4. Dodatek

4.1. Błędy numeryczne

Obliczenia wykonywane na komputerach mają skończoną dokładność, wynikającą z reprezentacji liczb w systemie komputerowym. Należy mieć świadomość, że pamięć komputera jest skończona, a w związku z tym w trakcie obliczeń numerycznych wykonywane są (najczęściej niejawnie!) zaokrąglenia i obcięcia, które, kumulując się w kolejnych iteracjach, mogą istotnie wpływać na wynik działania programu. Języki *C* oraz *C++* należą do języków **typowanych statycznie**. W związku z tym należy poświęcić szczególną uwagę na dobierane typów zmiennych, przewidując, w jakim zakresie liczbowym będzie zmieniać się wartość zmiennej (patrz: tabela 1 [Ćw. 2]). Jeżeli nie zostanie to dopilnowane, może dojść do **przekroczenia zakresu** danego typu. W przypadku liczb bez znaku (***unsigned char***, ***unsigned short***, ***unsigned int***, ...) przekroczenie zakresu powoduje **wyzerowanie zmiennej**. Dalsze zwiększanie wartości zmiennej będzie zachowywać się w standardowy sposób. Znajduje to zastosowanie m.in. w implementacji różnego rodzaju *liczników*. W przypadku liczb ze znakiem (***char***, ***short***, ***int***, ...) przekroczenie zakresu liczbowego skutkuje **niezdefiniowanym zachowaniem**. Nie jest sprecyzowane jaką wartość przyjmie wówczas zmienna. **Jest to zachowanie bardzo niebezpieczne dla działania aplikacji**. Jeżeli istnieje podejrzenie przekroczenia zakresu dla zmiennej typu znakowego, można się przed tym zabezpieczyć wykorzystując operator % (listing 15). Maksymalne i minimalne wartości dla poszczególnych typów zmiennych zostały zdefiniowane w pliku nagłówkowym **limits.h** biblioteki standardowej.

```
int x = INT_MAX; // 2147483647
// x++; // Niezdefiniowane zachowanie!
x = (x % INT_MAX) + 1; // x == 1
```

Listing 15. Zabezpieczenie przed przekroczeniem zakresu liczbowego

Zadanie (dla chętnych) Sprawdź, jak zachowują się zmienne o różnych typach po przekroczeniu zakresu liczbowego. Zabezpiecz operacje matematyczne takie, jak dodawanie czy odejmowanie przed przekroczeniem minimalnej i maksymalnej wartości, jaką może przyjąć zmienna.

Błędy numeryczne można podzielić na kilka rodzajów. Do najważniejszych należą:

- **błędy danych wejściowych**
- **błędy zaokrągleń** – np. reprezentacja nieskończonego rozwinięcia dziesiętnego $\frac{1}{3} = 0.(3)$
- **błędy obcięcia** – związane ze skończoną reprezentacją w pamięci komputera nieskończonych wyrażeń matematycznych, np. rozwinięcia w szereg

W kontekście obliczeń numerycznych ważnym pojęciem jest tak zwany ***epsilon maszynowy***. Jest to ***najmniejsza liczba nieujemna***, której dodanie do jedności da wynik różny od jedności (3). Definiuje on precyzję obliczeń numerycznych na liczbach zmiennoprzecinkowych.

$$1 + \epsilon \neq 1 \tag{3}$$

Wartość epsilona maszynowego może być wyznaczona w sposób przybliżony według następującego algorytmu:

1. przyjmij $\epsilon = 1$
2. sprawdź czy spełniony jest warunek $1 + \epsilon/2 == 1$

3. jeżeli warunek nie jest spełniony, to zapisz $\epsilon = \epsilon/2$ i przejdź do punktu 2, w przeciwnym wypadku ϵ to przybliżona wartość epsilona maszynowego

Zadanie (dla chętnych) Oblicz przybliżoną wartość epsilona maszynowego dla różnych typów zmiennoprzecinkowych (***float***, ***double***, ***long double***), a następnie porównaj otrzymane wyniki z wartościami zdefiniowanymi w pliku nagłówkowym **float.h** (***FLT_EPSILON***, ***DBL_EPSILON***, ***LDBL_EPSILON***).

Kumulacja błędów numerycznych może prowadzić do problemów przy porównywaniu liczb zmiennoprzecinkowych. Na listingu 16. przedstawiono kod programu obliczającego pole trójkąta równobocznego na podstawie znanych współrzędnych dwóch z jego wierzchołków, a następnie porównującego otrzymany wynik z wartością oczekiwaną. Funkcja ***area()*** przyjmuje kolejno współrzędne wierzchołka A (*x_1*, *y_1*) oraz współrzędne wierzchołka B (*x_2*, *y_2*) i oblicza pole trójkąta równobocznego korzystając ze wzorów:

a) długość boku trójkąta:

$$a = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{4}$$

b) pole trójkąta

$$P = \frac{a^2 \cdot \sqrt{3}}{4} \tag{5}$$

Na listingu 16. funkcja ***area()*** została wywołana dla następujących argumentów: A(1, 3) i B(6, 4). Podstawiając te wartości do równania (4) otrzyma się długość boku $a = |AB| = \sqrt{26}$, zatem oczekiwana wartość pola trójkąta wynosi $P = \frac{26 \cdot \sqrt{3}}{4}$.

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

double area(int x_1, int y_1, int x_2, int y_2) {
  double edge = sqrt(pow((x_1 - x_2), 2) + pow((y_1
   - y_2), 2));
  return pow(edge, 2) * sqrt(3) / 4;
}

int main() {
  double expectedValue = 26 * sqrt(3) / 4;
  double calculatedValue = area(1, 3, 6, 4);

  printf("Expected value: %.16f\n", expectedValue);
  printf("Calculated value: %.16f\n",
   calculatedValue);

  if (calculatedValue == expectedValue)
    printf("Area calculated correctly\n");
  else
    printf("Area calculated incorrectly\n");
  return 0;
}
```

Listing 16. Porównywanie liczb zmiennoprzecinkowych

Po skompilowaniu i uruchomieniu programu otrzymuje się następujący wynik:

```
Expected value: 11.2583302491977015
Calculated value: 11.2583302491976998
Area calculated incorrectly
```

Niezgodność wartości zwróconej z funkcji ***area()*** z wartością oczekiwaną wynika z przeprowadzenia ciągu operacji pierwiastkowania (obliczanie długości boku) i potęgowania (oblicznie pola trójkąta). Pierwiastek $\sqrt{26}$ jest liczbą niewymierną o nieskończonym rozwinięciu dziesiętnym, przez co jej reprezentacja w pamięci komputera jest obarczona **błędem zaokrąglenia**. Wynika z tego wniosek, że z powodu wystąpienia błędów numerycznych nie jest spełniona równość $\sqrt{26}^2 = 26$:

```
Sqrt(26)^2: 25.9999999999999964
```

Aby rozwiązać ten problem można zmodyfikować kod funkcji ***area()***, eliminując zbędną operację pierwiastkowania, lub zmieniając metodę porównywania liczb zmiennoprzecinkowych. Pierwsze rozwiązanie przedstawiono na listingu 17.

```
1 double area(int x_1, int y_1, int x_2, int y_2) {
2   double edgeSquared = pow((x_1 - x_2), 2) + pow((
    y_1 - y_2), 2);
3   return edgeSquared * sqrt(3) / 4;
4 }
```

Listing 17. Porównywanie liczb zmiennoprzecinkowych, eliminacja błędów zaokrąglenia

Wówczas, po skompilowaniu i uruchomieniu programu otrzyma się wynik:

```
Expected value: 11.2583302491977015
Calculated value: 11.2583302491977015
Area calculated correctly
```

Drugie podejście stanowi porównywanie liczb zmiennoprzecinkowych ze skończoną dokładnością. Operator porównania (==), zastosowany na listingu 16. (linia 16), zwraca wartość **true**, jeśli oba argumenty są równe dla

pełnej reprezentacji w pamięci komputera. Jeżeli chce się określić czy dwie zmienne są równe dla określonej tolerancji, to można posłużyć się funkcją przedstawioną na listingu 18. Funkcja ***fabs()***, zdeklarowana w pliku nagłówkowym **math.h**, zwraca wartość bezwzględną z liczby zmiennoprzecinkowej.

```
bool isEqual(double a, double b, double delta) {
  return (fabs(a - b) < delta);
}

\\ Przyklad uzycia
if (isEqual(calculatedValue, expectedValue, 1E-12))
  printf("Area calculated correctly\n");
else
  printf("Area calculated incorrectly\n");
```

Listing 18. Porównywanie liczb zmiennoprzecinkowych ze skończoną dokładnością

Literatura

[1] *C Operator Precedence*. URL: `https://en.cppreference.com/w/c/language/operator_precedence`.

[2] *C++ Operator Precedence*. URL: `https://en.cppreference.com/w/cpp/language/operator_precedence`.

[3] I. L. Chuang M. A. Nielsen. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010. ISBN: 978-1-107-00217-3.