Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Laboratorium informatyki

Ćwiczenie nr 4. Standardowe wejście i wyjście w językach *C* i *C++*

Zagadnienia do opracowania:

- funkcje wejścia i wyjścia w języku *C*
- obiektowe wejście i wyjście w języku *C++*
- przestrzenie nazw i ich zastosowanie
- typy wyliczeniowe
- instrukcja wielokrotnego wyboru *switch–case*

Spis treści

1. Cel ćwiczenia

Celem ćwiczenia jest zapoznanie z obsługą i różnicami między standardowym wejściem i wyjściem w językach *C* i *C++*.

2. Wprowadzenie

2.1. Funkcje wejścia i wyjścia w języku *C*

Podstawowymi funkcjami umożliwiającymi komunikację między użytkownikiem a programem są **funkcje wejścia/wyjścia**, których przedstawicielami w języku *C* są odpowiednio funkcja ***scanf()*** i ***printf()***. Obie funkcje są zdeklarowane w pliku ***stdio.h***, a ich nagłówki przedstawiono na listingu 1. Obie funkcje jako argumenty przyjmują ***literał łańcuchowy*** – napis, reprezentowany w języku *C/C++* przez *wskaźnik na typ char* (*const char **) *[Więcej informacji na temat wskaźników w Ćw. 6]* i ***zmienną listę argumentów***, zapisywaną w postaci wielokropka. Funkcje zwracają liczbę całkowitą, która dla funkcji:

- ***printf()***, w przypadku powodzenia, stanowi sumaryczną liczbę zapisanych znaków, albo jest wartością ujemną, sygnalizującą niepowodzenie;
- ***scanf()***, w przypadku powodzenia, stanowi liczbę poprawnie zapisanych zmiennych z listy argumentów, albo jest wartością ujemną (makrodefinicja EOF – *End–of–File*), oznaczającą, że nastąpił błąd odczytu lub natrafiono na symbol *EOF*.

```
1 int printf(const char * format, ...);
2 int scanf(const char * format, ...);
```

Listing 1. Nagłówki funkcji wejścia/wyjścia w języku *C*

Funkcje *printf()* i *scanf()* są przykładami funkcji o **nieokreślonej liczbie argumentów** (*ang. variadic functions*). Zmienna lista argumentów może być reprezentowana zarówno przez zero, jeden czy więcej argumentów, z którymi wywołana będzie funkcja. Jeżeli przekazany do funkcji łańcuch znaków zawiera tzw. ***specyfikator formatu*** (*%d, %f, %c, ...*), to zgodnie z określonym formatem przetwarzany jest kolejny argument funkcji z listy. W przypadku funkcji *printf()* kolejny argument z listy zostanie odpowiednio sformatowany i wklejony do napisu, w miejsce odpowiadającego mu specyfikatora. Po obsłużeniu wszystkich argumentów przetworzony łańcuch znakowy zostanie wpisany do ***standardowego wyjścia*** (*stdout*). Najczęściej strumień standardowego wyjścia przekierowywany jest na konsolę systemową. Przykłady wywołań funkcji *printf()* z różną liczbą argumentów przedstawiono na listingu 2.

```
1  // Pusta zmienna lista argumentow
2  printf("Hello, world!\n");
3
4  // Jeden argument dodatkowy
5  printf("Number passed explicitly: %d\n", 666);
6  int x = 24;
7  printf("Number passed via value: %d\n", x);
8
9  // Wiecej argumentow
10 printf("Different value types: %d, %f, %c\n", 125,
     2.0, 'A');
```

Listing 2. Wywołanie funkcji *printf()* z różną liczbą argumentów

Standard definiuje następujący prototyp *specyfikatora formatu* dla funkcji ***printf()***:

%[flagi][szerokość][.precyzja][długość]symbol

Za pomocą *flagi* można określić sposób wyświetlania wartości przekazanej do funckji *printf()*. *Szerokość* określa minimalną liczbę znaków, jaka ma zostać wyświetlona. *Precyzja* umożliwia ustalenie liczby cyfr do wyświetlenia. *Symbol* stanowi rodzaj zmiennych przekazywanych jako argument (dla zmiennych liczbowych określa m.in. ich bazę). *Długość* mówi za pomocą jakiego typu zmiennej ma być interpretowany przekazany argument, dla danego *symbolu* specyfikatora. Wszystkie parametry specyfikatora, poza *symbolem*, są nieobowiązkowe w użyciu. W tabeli 1. zestawiono wybrane specyfikatory formatu funkcji *printf()*. Pełna lista dostępna jest pod adresem `http://www.cplusplus.com/reference/cstdio/printf/`. Przykład wywołania funkcji *printf()* z różnymi specyfikatorami formatu przedstawiono na lisintgu 3. Operator & pobiera adres z pamięci komputera, pod którym została zaalokowana dana zmienna (tu: &*buffer*).

Tabela 1. Zestawienie wybranych specyfikatorów formatu funkcji *printf()* [2]

flagi	**Opis**
–	wyrównaj do lewej w obrębie zadanej szerokości (domyślnie wyrównanie do prawej)
+	wyświetl znak przy liczbie (domyślnie tylko dla liczb ujemnych)
(spacja)	wstaw pojedynczą spację przed wartością (w przypadku niewyświetlania znaku)
#	wyświetl bazę liczbową (0, 0x, 0X) przed wartością (przy zastosowaniu odpowiednio specyfikatorów **o, x, X**)
0	poprzedź wartość zerami w obrębie zadanej szerokości (domyślnie poprzedzana spacjami)
szerokość	**Opis**
(liczba)	minimalna liczba znaków, jaka ma zostać wyświetlona (dopełnij wartość do zadanej szerokości poprzedzającymi spacjami/zerami)
.precyzja	**Opis**

.liczba	dla specyfikatorów **a, A, e, E, f, F**: liczba cyfr po przecinku do wyświetlenia dla specyfikatorów **d, i, o, u, x, X**: liczba cyfr do wyświetlenia (dopełnij wartość do zadanej precyzji poprzedzającymi spacjami/zerami)

długość	**Opis**
(brak)	*int* (dla symboli ***d, i, c***) *unsigned int* (dla symboli ***u, o, x, X***) *double* (dla symboli ***f, F, e, E, g, G, a, A***) *char ** (dla symbolu ***s***) *void ** (dla symbolu ***p***)
hh	*char* (dla symboli ***d, i***) *unsigned char* (dla symboli ***u, o, x, X***)
h	*short* (dla symboli ***d, i***) *unsigned short* (dla symboli ***u, o, x, X***)
l	*long* (dla symboli ***d, i***) *unsigned long* (dla symboli ***u, o, x, X***) *wint_t*[1] (dla symbolu ***c***) *wchar_t ** (dla symbolu ***s***)
ll	*long long* (dla symboli ***d, i***) *unsigned long long* (dla symboli ***u, o, x, X***)
L	*intmax_t*[2] (dla symboli ***d, i***) *long double* (dla symboli ***f, F, e, E, g, G, a, A***)

symbol	**Opis**
d / i	dziesiętna liczba całkowita ze znakiem
u	dziesiętna liczba całkowita bez znaku
o	ósemkowa liczba całkowita bez znaku
x	szesnastkowa liczba całkowita bez znaku (minuskuły)
X	szesnastkowa liczba całkowita bez znaku (wersaliki)

[1] *wide int*; alias dla *wchar_t* lub *int*

[2] typ całkowity ze znakiem o największym wspieranym rozmiarze

f	dziesiętna liczba zmiennoprzecinkowa (minuskuły)
F	dziesiętna liczba zmiennoprzecinkowa (wersaliki)
e	notacja wykładnicza liczby zmiennoprzecinkowej (minuskuły)
E	notacja wykładnicza liczby zmiennoprzecinkowej (wersaliki)
g	krótsza reprezentacja z %e lub %f
G	krótsza reprezentacja z %E lub %F
a	szesnastkowa liczba zmiennoprzecinkowa (minuskuły)
A	szesnastkowa liczba zmiennoprzecinkowa (wersaliki)
c	znak (liczba, litera, symbol, ...)
s	łańcuch znakowy
p	adres (wskaźnik)
%	%

```
// %[#][][][]X
printf("%#X\r\n", 0xCAFEBABE); // 0XCAFEBABE
// %[0][7][][]d
printf("%07d\r\n", 666); // 0000666
// %[][][][ll]d
printf("%lld\r\n", 112LL); // 112
// %[+][][.3][]f
printf("%+.3f\r\n", 1.25); // +1.250
// %[][][][]s
printf("%s\r\n", "text"); //text
// %[][][][]%
printf("%%\r\n"); // %
```

Listing 3. Wywołanie funkcji *printf()* z różnymi specyfikatorami formatu

Programując w języku *C* można spotkać się z różnymi symbolami, oznaczającymi przejście do nowej linii:

- **\r** (*Carriage Return, CR*) – *powrót karetki*, używany w programach pisanych na systemy operacyjne Mac OS, przed wersją X
- **\n** (*Line Feed, LF*) – używany w programach pisanych na systemy operacyjne Unix/Mac OS X
- **\r\n** (*CR + LF*) – używany w programach pisanych na systemy operacyjne Windows

Obsługa ***standardowego wejścia*** w języku *C* realizowana jest z wykorzystaniem funkcji ***scanf()***. Jeżeli przekazany do funkcji łańcuch znaków zawiera ***specyfikator formatu***, to zgodnie z określonym formatowaniem do kolejnego argumentu z listy zostanie wpisana wartość, pobrana ze ***standardowego wejścia*** (*stdin*). Funkcja ***scanf()*** operuje na ***adresach zmiennych***, dlatego też przekazując argumenty do funkcji należy posłużyć się ***operatorem adresu*** (&). Przykład wywołania funkcji *scanf()* przedstawiono na listingu 4.

```
1 int number;
2 scanf("%d", &number);
```

Listing 4. Wywołanie funkcji *scanf()*

Standard definiuje następujący prototyp *specyfikatora formatu* dla funkcji ***scanf()***:

%[*][szerokość][długość]symbol

Opcjonalny znak * określa, że zaczytana ze standardowego wejścia wartość ma zostać pominięta (nie będzie zapisana pod kolejnym adresem, przekazanym przez listę argumentów). *Szerokość* określa maksymalną liczbę znaków, jaka może zostać odczytana w pojedynczej operacji. *Symbol* i *długość* pełnią analogiczną rolę, jak w przypadku funkcji *printf()* (patrz: tabela 1). Szczegółowe informacje można odnaleźć w referencji `http://www.`

cplusplus.com/reference/cstdio/scanf/. Przykład wywołania funkcji *scanf()* z różnymi specyfikatorami formatu przedstawiono na lisintgu 5.

```
1 // %[][8][l]x
2 unsigned long number;
3 scanf("%8lx", &number); // FA2B3C12DABC -> number =
     FA2B3C12
4 // %[][][]d %[*][][]c
5 int number;
6 scanf("%d%*c", &number); // 123f -> number = 123
```

Listing 5. Wywołanie funkcji *scanf()* z różnymi specyfikatorami formatu

Literały liczbowe, reprezentujące określone wartości zmiennych, w językach *C* i *C++* mogą posiadać różnego rodzaju przedrostki i przyrostki. Przedrostki mogą określać bazę wybranej reprezentacji liczby:

- **brak przedrostka** – liczba w systemie dziesiętnym
- **0b** – liczba w systemie binarnym (od *C++*14)
- **0** – liczba w systemie ósemkowym
- **0x, 0X** – liczba w systemie szesnastkowym

Za pomocą przyrostków można określać typ literału liczbowego, np. jawnie przekazując zadaną wartość jako parametr wywołania funkcji:

- **brak przyrostka** – *int* (dla liczb całkowitych), *double* (dla liczb zmiennoprzecinkowych)
- **u/U** – *unsigned int*
- **l/L** – *long*
- **ll/LL** – *long long*

- **u/U + l/L** – *unsigned long*
- **u/U + ll/LL** – *unsigned long long*
- **f** – *float*

```
func(2.0); // double
func(2.3f); // float
func(44); // int
func(12u); // unsigned int
func(123456ULL); // unsigned long long
```

2.2. Obiektowe wejście i wyjście w języku *C++*

Jednym z założeń języka *C++* jest **wsteczna kompatybilność**, zarówno w odniesieniu do poprzednich standardów języka, ale również w kontekście języka *C* (poza kilkoma wyjątkami). Dlatego też, pomimo występowania w języku *C++* obiektowego odpowiednika mechanizmu obsługi wejścia i wyjścia, możliwe jest korzystanie z funkcji *printf()* i *scanf()*, na tych samych zasadach jak w języku *C*. Dodatkowo obsługa standardowego wejścia i wyjścia, w języku *C++* realizowana jest za pomocą, odpowiednio, obiektów ***std::cin*** i ***std::cout***. Przekazywanie argumentów do strumienia wejściowego i wyjściowego odbywa się z wykorzystaniem przeciążanych operatorów przesunięcia bitowego, odpowiednio ***operatora »*** oraz ***operatora «*** *[Więcej informacji o przeciążaniu operatorów w Ćw. 5]*. Przykład zastosowania obiektów ***std::cout*** i ***std::cin*** przedstawiono na listingu 6. Obiekt ***std::endl*** realizuje przejście do nowej linii w sposób uniwersalny (niezależny od systemu operacyjnego). *Uwaga: aby skompilować poniższy program należy posłużyć się kompilatorem języka C++, np. kompilatorem* ***g++***.

```cpp
#include <iostream>

int main() {
    std::cout << "Podaj liczbe calkowita: ";
    int number;
    std::cin >> number;
    std::cout << std::endl << "Podano liczbe: " <<
   number << std::endl;
    return 0;
}
```

Listing 6. Zastosowanie obiektów *std::cout* i *std::cin*

Przeciążone operatory przesunięcia bitowego umożliwiają tzw. ***wywołania łańcuchowe***. Oznacza to, że wywołania:

```cpp
float number = 14.46;
std::cout << number;
std::cout << "napis";
std::cout << std::endl;
std::cout << 666;
```

można zastąpić ciągiem:

```cpp
float number = 14.46;
std::cout << number << "napis" << std::endl << 666;
```

2.3. Przestrzenie nazw

Podczas pracy nad dużymi projektami, jak i korzystając z wielu ***bibliotek*** zewnętrznych, można napotkać problem ***konfliktu nazw***, występujący gdy różne funkcje o tej samej sygnaturze, zmienne bądź stałe o tym samym zasięgu posiadają te same nazwy. Język *C++* wprowadza udogodnienie, umożliwiające rozwiązanie tego problemu w postaci ***przestrzeni nazw*** (*namespace*). ***Przestrzenie nazw*** stanowią ***obszary deklaracyjne*** dla funkcji, stałych i zmiennych. Elementy zdeklarowane w jednej przestrzeni nazw nie kolidują z elementami zdeklarowanymi w innej przestrzeni, nawet jeśli posiadają tę samą nazwę. Do definiowania przestrzeni nazw wykorzystuje się słowo kluczowe ***namespace***. Zmienne czy funkcje zdeklarowane poza **jawnie określonymi przestrzeniami nazw** należą do tzw. ***globalnej przestrzeni nazw***. Aby odwołać się do elementów przestrzeni nazw należy posłużyć się operatorem rozdzielczości zakresu (***operator ::***):

```
1  namespace myspace {
2      int number = 1;
3  }
4  namespace otherspace {
5      // Brak kolizji nazwy ze zmienna number z
       przestrzeni myspace
6      int number = 2;
7  }
8  // Globalna przestrzen nazw
9  int number = 3;
10
11 // Wyswietl 3
12 std::cout << number << std::endl;
13 // Wyswietl 1
14 std::cout << myspace::number << std::endl;
```

Przestrzenie nazw można zagnieżdżać (np. w celu rozwiązania konfliktów nazw wewnątrz poszczególnych przestrzeni nazw):

```
// Rownowaznie namespace first::second::third { int number = 2; }
namespace first {
    namespace second {
        namespace third {
            int number = 2;
        }
    }
}

std::cout << first::second::third::number;
```

Obiekty biblioteki standardowej, jak ***cout*** czy ***cin*** zostały zdeklarowane w przestrzeni nazw ***std***. Odwołując się do nich należy wykorzystać operator rozdzielczości zakresu (*std::cout*, *std::cin*) lub posłużyć się ***deklaracją*** lub ***dyrektywą using***. Służą one do **importowania nazw z określonej przestrzeni nazw do danego obszaru deklaracyjnego**. Innymi słowy, w obrębie określonego zakresu (bloku, jednostki translacji) widoczna będzie nazwa, pochodząca z obcej przestrzeni nazw tak, jakby została zdeklarowana w danej przestrzeni:

```
#include <iostream>

void printVar(int x) {
    using std::cout;
    cout << x;
}

```

```
8  int main() {
9      using namespace std;
10     int number;
11     cin >> number;
12     printVar(number);
13     return 0;
14 }
```

Deklaracja using (*using std::cout*) importuje **pojedynczą nazwę** z zadanej przestrzeni nazw (tu: obiekt *cout*) w obrębie zadanego bloku (tu: w obrębie funkcji *printVar()*). ***Dyrektywa using*** (*using namespace std*) importuje **wszystkie nazwy** z zadanej przestrzeni nazw w obrębie zadanego bloku (tu: funkcji *main()*). Importowanie nazw może prowadzić do ich ***kolizji*** z nazwami już występującymi w zadanym obszarze deklaracyjnym czy nazwami importowanymi z innych bibliotek. Dlatego też **odradza się stosowania dyrektywy using, na rzecz deklaracji lub jawnego odwoływania się do nazw z wykorzystaniem operatora rozdzielczości zakresu. W szczególności operator rozdzielczości zakresu powinien być używany w plikach nagłówkowych. Dobra praktyka wymaga, żeby w plikach nagłówkowych nie wykorzystywać dyrektywy using w ogóle.**

Język *C++* wprowadza jeszcze jeden rodzaj przestrzeni nazw – ***anonimowe przestrzenie nazw***. Są to nienazwane przestrzenie, a wszystkie funkcje czy zmienne zdeklarowane wewnątrz tych przestrzeni zachowują się tak, jakby były zdeklarowane z użyciem słowa kluczowego ***static***. Ponieważ przestrzenie te nie posiadają nazwy, nie można odwołać się do nich z wykorzystaniem ***deklaracji*** lub ***dyrektywy using***. Wszystkie elementy anonimowych przestrzeni nazw mają widoczność w obrębie jednostki translacji:

```cpp
// Anonimowa przestrzen nazw
namespace {
    // Rownowaznie static float number = 2.0;
    float number = 2.0;

    // Rownowaznie static void print(float number)
    void print(float number) {
        std::cout << number << std::endl;
    }
}
```

2.4. Typy wyliczeniowe

Typ wyliczeniowy, to rodzaj typu danych, którego wartości opisane są za pomocą ***literałów wyliczeniowych***, zwanych ***enumeratorami***. Z jego pomocą reprezentuje się w kodzie programu zmienne, które trudno w sposób oczywisty wyrazić za pomocą liczb (np. barwy, dni tygodnia, marki samochodów). Typy wyliczeniowe definiowane są z wykorzystaniem słowa kluczowego ***enum***. Przykład zaprezentowano na listingu 7. ***Colour*** stał się nowym typem wyliczeniowym, którego zbiór wartości reprezentowany jest przez enumeratory {***RED***, ***BLUE***, ***GREEN***, ***YELLOW***}. Przyjęło się zapisywać nazwę typu wyliczeniowego wielką literą, a literały wyliczeniowe kapitalikami.

```
enum Colour {
    RED,
    BLUE,
    GREEN,
    YELLOW
};

// Przypisanie wartosci BLUE do zmiennej someColour
   typu Colour
Colour someColour = BLUE;
```

Listing 7. Definiowanie typów wyliczeniowych

Poszczególne literały wyliczeniowe stanowią stałe symboliczne, odpowiadające kolejnym liczbom całkowitym. Domyślnie, pierwsza zdefiniowana wartość odpowiada liczbie 0. Dlatego też możliwe jest następujące przypisanie:

```
// colourValue == 2
int colourValue = GREEN;
```

Możliwe jest jawne przypisywanie wartości enumeratorom, np.:

```
enum Day {
    MONDAY = 1,
    WEDNESDAY = 3,
    FRIDAY = 5,
    SATURDAY = 6
};
```

Początkowo język *C* dopuszczał przypisywanie (ze zdefiniowanym rezultatem) do zmiennych wyliczeniowych jedynie wartości odpowiadających zdefiniowanym literałom wyliczeniowym. Jednakże język *C++* rozszerzył tę za-

sadę na wartości leżące między najmniejszą a największą wartością odpowiadającą danym literałom wyliczeniowym:

```
// Poprawne przypisanie
Day someDay = Day(4);
// Niezdefiniowany rezultat (liczba poza zakresem 1-6)
Day anotherDay = Day(7);
```

Standard *C++11* wprowadził nowe typy wyliczeniowe, definiowane z wykorzystaniem słów kluczowych ***enum class*** (lub równoważnie ***enum struct***). Posiadają one swój zakres widoczności, dlatego, w przeciwieństwie do typu ***enum*** z języka *C*, odwołując się do poszczególnych enumeratorów należy posłużyć się operatorem rozdzielczości zakresu (***operator ::***). Charakteryzują się one szeregiem zalet w stosunku do typów wyliczeniowych języka *C*:

1. możliwe jest używanie tych samych enumeratorów w obrębie różnych typów wyliczeniowych (klasyczny ***enum*** z języka *C* wymusza unikatowość literałów wyliczeniowych);

```
enum class CarPaint {
    BLUE,
    SILVER,
    BLACK
};

enum class NailPolishColour {
    RED,
    BLACK, // Blad w przypadku enum (wspolny enumerator z CarPaint)
    YELLOW
};
```

2. zmienne mogą posiadać identyfikatory (nazwy) takie, jak literały wyliczeniowe (niemożliwe w przypadku typów wyliczeniowych z języka C);

```
// Blad w przypadku enum (identyfikator
   zarezerwowany dla enumeratora CarPaint)
int BLUE;
```

3. nie jest możliwe porównanie enumeratorów zdefiniowanych w różnych typach wyliczeniowych, jak również przypisanie enumeratora do typu całkowitego, bez jawnego rzutowania.

```
// true w przypadku enum, blad w przypadku enum
   class
if (CarPaint::BLUE == NailPolishColour::RED) {
    ...
}

// OK w przypadku enum, blad w przypadku enum
   class
int colourValue = CarPaint::BLACK;

// OK w przypadku enum class (jawne rzutowanie
   na typ int)
int colourValue = static_cast<int>(CarPaint::
   BLACK);
```

Możliwe jest również określanie typu całkowitego powiązanego ze stałymi symbolicznymi w obrębie typu wyliczeniowego:

```
enum class FastFood : unsigned short {
    HAMBURGER = 22,
    FRIES = 8,
    COLA = 7
};

unsigned short price = static_cast<unsigned short>(
   FastFood::HAMBURGER);
```

2.5. Więcej o instrukcjach warunkowych

Instrukcja ***if–else*** nie jest jedyną **instrukcją warunkową** dostępną w języku *C/C++*. Często alternatywnym wyborem może być ***instrukcja wielokrotnego wyboru switch–case***. Jej działanie polega na warunkowym skoku programu do odpowiedniej etykiety (***case***), której wartość odpowiada przekazanemu do instrukcji ***switch*** **wyrażeniu całkowitemu**. Etykiety mogą stanowić zmienne typu całkowitego (jak *int* lub *char*) czy ***typy wyliczeniowe***. Jeżeli wyrażenie całkowite nie jest równe co do wartości żadnej z etykiet, program wykona skok warunkowy do słowa kluczowego ***default***. Przykład zastosowania instrukcji ***switch–case*** przedstawiono na listingu 8. Jest to przykład aplikacji realizującej prosty kalkulator (dodawanie lub odejmowanie). Na wejściu użytkownik proszony jest o podanie dwóch liczb całkowitych, a następnie określenie operacji arytmetycznej, reprezentowanej przez odpowiadające im wartości 0 oraz 1. Liczby te stanowią etykiety instrukcji ***switch–case***. Jeżeli użytkownik wpisze wartość 0 – wykonany zostanie skok do etykiety ***case 0***, a liczby całkowite zostaną zsumowane, jeżeli wartość 1 – liczby te zostaną odjęte od siebie. Jeżeli użytkownik postanowi podać inną wartość, odpowiadającą nieobsługiwanej operacji arytmetycznej, program wykona skok do etykiety ***default***, a użytkownik zostanie poinformowany, że wybrał nieokreślone działanie. Słowo kluczowe ***break*** służy przerwaniu instrukcji wielokrotnego wyboru ***switch–case***. W przypadku

braku instrukcji ***break***, po skoku do etykiety ***case 1*** wykonałyby się również wszystkie pozostałe etykiety, łącznie z ***default***.

```
#include <iostream>

int main() {
    int x, y, result = 0;
    unsigned int condValue;
    std::cout << "Podaj dwie liczby calkowite." << std::endl << "x: ";
    std::cin >> x;
    std::cout << std::endl << "y: ";
    std::cin >> y;
    std::cout << std::endl << "Wybierz 0 - aby dodac liczby, 1 - aby je odjac: ";
    std::cin >> condValue;
    std::cout << std::endl << "Wybrano: " << condValue << std::endl;

    switch (condValue) {
        case 0:
            result = x + y;
            break;
        case 1:
            result = x - y;
            break;
        default:
            std::cout << "Nieokreslone dzialanie" << std::endl;
    }

```

```
25      std::cout << "Wynik operacji na liczbach: " <<
    result << std::endl;
26      return 0;
27  }
```

Listing 8. Zastosowanie instrukcji wielokrotnego wyboru *switch–case*

Na listingu 9. pokazano zmodyfikowaną wersję aplikacji kalkulatora z wykorzystaniem typu wyliczeniowego. Aby umożliwić konwersję wartości liczbowych z całego dostępnego zakresu na typ wyliczeniowy *Operation*, dodano enumerator *INVALID* o wartości równej maksymalnej wartości możliwej do zapisania pod zmienną typu *unsigned int* (tu: *condValue*). Warto zwrócić uwagę, że typ całkowity powiązany ze stałymi symbolicznymi typu *Operation* został w sposób jawny określony jako *unsigned int*. Wywołanie ***std::numeric_limits<unsigned int>::max()*** stanowi odpowiednik w języku *C++* dla makra ***UINT_MAX*** z języka *C*, określającego maksymalną wartość zmiennej typu *unsigned int*.

```
1   #include <iostream>
2   #include <limits>
3
4   enum class Operation : unsigned int {
5       ADD = 0,
6       SUBTRACT = 1,
7       // Aby zapewnic zdefiniowane zachowanie dla
    wartosci wiekszych niz 1
8       INVALID = std::numeric_limits<unsigned int>::max
    ()
9   };
10
11  int main() {
12      int x, y, result = 0;
```

```
13      unsigned int condValue;
14      std::cout << "Podaj dwie liczby calkowite." << std::endl << "x: ";
15      std::cin >> x;
16      std::cout << std::endl << "y: ";
17      std::cin >> y;
18      std::cout << std::endl << "Wybierz 0 - aby dodac liczby, 1 - aby je odjac: ";
19      std::cin >> condValue;
20      std::cout << std::endl << "Wybrano: " << condValue << std::endl;
21      Operation operation = static_cast<Operation>(condValue);
22      switch (operation) {
23          case Operation::ADD:
24              result = x + y;
25              break;
26          case Operation::SUBTRACT:
27              result = x - y;
28              break;
29          default:
30              std::cout << "Nieokreslone dzialanie" << std::endl;
31      }
32
33      std::cout << "Wynik operacji na liczbach: " << result << std::endl;
34      return 0;
35  }
```

Listing 9. Zastosowanie typów wyliczeniowych w instrukcji wielokrotnego wyboru *switch–case*

Oprócz instrukcji warunkowych, zarówno język *C* jak i *C++* posiadają ***trójargumentowy operator wyboru ?:***. Jego składnia wygląda następująco:

warunek ? wyrażenie_1 : wyrażenie_2

Jeżeli ***warunek*** jest prawdą (ma logiczną wartość ***true*** – różną niż 0), to wartością zwracaną przez ***operator?:*** jest ***wyrażenie_1***. W przeciwnym wypadku operator zwraca ***wyrażenie_2***. Przykład zastosowania ***operatora ?:*** w aplikacji z listingu 8. przedstawiono na listingu 10.

```
1  #include <iostream>
2
3  int add(int x, int y) {
4      return x + y;
5  }
6
7  int subtract(int x, int y) {
8      return x - y;
9  }
10
11 int main() {
12     int x, y;
13     unsigned int condValue;
14     std::cout << "Podaj dwie liczby calkowite." <<
       std::endl << "x: ";
15     std::cin >> x;
16     std::cout << std::endl << "y: ";
17     std::cin >> y;
18     std::cout << std::endl << "Wybierz 0 - aby dodac
       liczby, 1 - aby je odjac: ";
19     std::cin >> condValue;
```

```
20      std::cout << std::endl << "Wybrano: " <<
    condValue << std::endl;
21
22      if (condValue != 0 && condValue != 1) {
23          std::cout << "Nieokreslone dzialanie" << std
    ::endl;
24      } else {
25          int result = (condValue == 0) ? add(x, y) :
    subtract(x, y);
26          std::cout << "Wynik operacji na liczbach: "
    << result << std::endl;
27      }
28
29      return 0;
30  }
```

Listing 10. Zastosowanie *operatora ?:*

3. Program ćwiczenia

Zadanie 1. Korzystając z referencji do biblioteki ***iomanip*** (`http://www.cplusplus.com/reference/iomanip/`) i posługując się obiektem ***std::cout***:

1. wyświetl liczbę w reprezentacji szesnastkowej, łącznie z bazą (0x)
2. ustaw szerokość pola (analogicznie do ***szerokości*** w specyfikatorze funkcji ***printf()***), a wyświetlaną wartość liczbową poprzedź dopełniającymi zerami (do ustalonej szerokości)
3. wyświetl liczbę zmiennoprzecinkową z określoną precyzją (liczbą cyfr po przecinku)

Zadanie 2. W pliku nagłówkowym ***calculator.h*** zawarto deklaracje czterech funkcji realizujących odpowiednio dodawanie, odejmowanie, mnożenie i dzielenie na liczbach zmiennoprzecinkowych oraz funkcję ***float calculate(Operation, float, float)***, która na podstawie typu operacji arytmetycznej (***enum class Operation***), wykonuje odpowiednie działanie matematyczne na dwóch liczbach zmiennoprzecinkowych (wywołuje jedną z czterech funkcji: ***add()***, ***subtract()***, ***multiply()*** lub ***divide()***) i zwraca wynik tej operacji. W pliku ***main.cpp*** zaimplementowano program prostego kalkulatora, który pobiera ze standardowego wejścia dwie liczby zmiennoprzecinkowe oraz typ operacji arytmetycznej i wywołuje funkcję ***calculate()***. W ramach zadania, w pliku ***calculator.cpp***:

1. zdefiniuj funkcje ***add()***, ***subtract()***, ***multiply()*** oraz ***divide()***. *Uwaga: w przypadku dzielenia przez zero wywołaj funkcję quitWithError(), zdefiniowaną w pliku* ***calculator.cpp****.*
2. zdefiniuj funkcję ***calculator*** tak, aby cały program działał poprawnie. *Uwaga: w przypadku wyboru błędnej operacji arytmetycznej wywołaj funkcję quitWithError(), zdefiniowaną w pliku* ***calculator.cpp****.*

Zadanie 3. Zmodyfikuj kod programu z zadania 2 (funkcję ***main()***) w ten sposób, aby program pobierał i przetwarzał liczby przekazane od użytkownika tak długo, aż użytkownik dwa razy z rzędu nie poda tej samej kombinacji liczb.

4. Dodatek

4.1. Nazewnictwo plików nagłówkowych

Już na samym początku pracy z językiem *C++* można zauważyć pewną nieścisłość dotyczącą konwencji nazewnictwa plików nagłówkowych. Osoby zaznajomione z językiem *C* wiedzą, że przyjęło się, aby pliki nagłówkowe posiadały rozszerzenie ***.h***, np. ***stdio.h***. Poznając standardowe wejście i wyjście w języku *C++* korzysta się z pliku nagłówkowego ***iostream***. W tym miejscu pojawia się niespójność. Plik nagłówkowy nie posiada żadnego rozszerzenia. Wraz z rozwojem języka zmieniono konwencję dotyczącą nazewnictwa plików nagłówkowych. I tak, w nowych standardach języka *C++* pliki nagłówkowe nie posiadają żadnego rozszerzenia (dotyczy to plików biblioteki standardowej; w prywatnych projektach najczęściej nadal spotyka się z rozszerzeniami ***.h/.hpp***). Pliki nagłówkowe biblioteki standardowej języka *C* zaadaptowane do języka *C++* odrzuciły rozszerzenie ***.h*** na rzecz przedrostka ***c***, np. ***time.h*** $\longrightarrow$ ***ctime***. Konwencję nazewnictwa plików nagłówkowych w językach *C* i *C++* zestawiono w tabeli 2.

Tabela 2. Konwencja nazewnictwa plików nagłówkowych w języku *C* i *C++*[1]

Rodzaj nagłówka	***Konwencja***
Styl *C*	rozszerzenie ***.h***
Stary styl *C++*	rozszerzenie ***.h***
Nowy styl *C++*	brak rozszerzenia
Konwersja z *C*	brak rozszerzenia, przedrostek ***c***

Literatura

[1] S. Prata. *Język C++. Szkoła programowania*. 6th ed. Helion, 2013. ISBN: 978-83-246-4336-3.

[2] *printf*. URL: `http://www.cplusplus.com/reference/cstdio/printf/`.