# Podręcznik do C/C++
# Marek Pudełko
# IV TI

# Spis treści

## Zawartość


Spis treści ...... 2
Pierwsze kroki w C/C++ ...... 5
  Edytor Dev-C++ dla początkujących ...... 5
  Pierwszy program w C/C++ ...... 8
  Budowa programu w C/C++ ...... 9
    Biblioteki ...... 9
    Przestrzeń nazw ...... 9
    Funkcja główna ...... 9
    Wykonywane polecenia ...... 10
  Polecenie cout ...... 10
  Zmienne w języku C/C++ ...... 11
  Stałe w języku C/C++ ...... 11
  Nazwy w języku C/C++ ...... 12
    Słowa kluczowe ...... 12
    Notacja węgierska ...... 12
Notacja węgierska ...... 12
Operatory w C/C++ ...... 17
  Operatory wieloargumentowe ...... 17
  Operatory zmiany znaku ...... 17
  Operatory matematyczne ...... 18
  Operatory krementacji ...... 19
  Operatory relacyjne ...... 20
  Operatory logiczne ...... 21
  Operatory bitowe ...... 21
  Operator warunkowy ...... 21
  Operatory wskaźnikowe ...... 21
  Operator rozmiaru ...... 22
  Operator wiązania ...... 22
  Operatory wyboru pola ...... 22
  Operatory nawiasowe ...... 23
  Priorytety operatorów ...... 23
Proste obliczenia matematyczne ...... 24
  Zapis wybranych konstrukcji matematycznych ...... 24
  Ćwiczenia ...... 25
Funkcje matematyczne ...... 26
  Popularne funkcje matematyczne ...... 26
  Funkcje trygonometryczne ...... 28
  Funkcje potęgowe i logarytmiczne ...... 30
  Ćwiczenia – funkcje trygonometryczne ...... 31
  Ćwiczenia – funkcje logarytmiczne ...... 32

Ćwiczenia różne ........................................................................................................ 32
Instrukcja IF ........................................................................................................ 35
Warunki instrukcji IF ........................................................................................................ 35
Operator warunkowy ........................................................................................................ 37
Zagnieżdżone instrukcje IF ........................................................................................................ 37
Operatory logiczne ........................................................................................................ 38
Ćwiczenia ........................................................................................................ 40
Instrukcja SWITCH..CASE ........................................................................................................ 41
Ćwiczenia ........................................................................................................ 42
Pętle w C/C++ ........................................................................................................ 43
Pętla FOR ........................................................................................................ 43
Pętla nieskończona ........................................................................................................ 44
Wyjście z pętli nieskończonej ........................................................................................................ 45
Wielokrotne pętle FOR ........................................................................................................ 46
Pętla WHILE ........................................................................................................ 48
Pętla warunkowa ........................................................................................................ 49
Pętla nieskończona ........................................................................................................ 50
Wyjście z pętli nieskończonej ........................................................................................................ 50
Wielokrotne pętle WHILE ........................................................................................................ 51
Pętla DO..WHILE ........................................................................................................ 52
Pętla warunkowa ........................................................................................................ 54
Ćwiczenia ........................................................................................................ 55
Funkcje matematyczne ........................................................................................................ 69
Dodatek A – zestawienie podstawowych funkcji C/C++ ........................................................................................................ 74

Wprowadzenie do języka C/C++

Historia języka C/C++

# Pierwsze kroki w C/C++

Chciałbym programować w C++, ale nie wiem od czego zacząć.

Czy ten mieliście (lub macie) takie dylematy? Sprawa nie jest na szczęście taka trudna. Potrzeba tak naprawdę kilku rzeczy.

1. Przede wszystkim musza to być chęci i wytrwałość. Nauka programowania na początku może sprawiać trudność i wiele osób po kilku tygodniach zniechęca się an dobre. A szkoda!

2. Kolejna rzecz to jakiś podręcznik. W księgarniach jest dość duży wybór pozycji zarówno dla początkujących jak i dla zaawansowanych. Można też się posłużyć kursami dostępnymi w Internecie.

3. Edytor C++, czyli program do pisania programów w tym języku. Jest ich bardzo wiele – zarówno płatne jak darmowe. Z profesjonalnych narzędzi polecam Borland C++ Builder. Ułatwia łatwe tworzenie aplikacji graficznych. Jednak początkujący używają najczęściej edytora Dev-C++. To proste środowisko programistyczne (i w dodatku darmowe). Jest spotykany praktycznie w każdej szkole i uczelni, co dowodzi jego prostoty i jednocześnie sporych możliwości. Popularność jest tym zaskakująca, ze program nie jest rozwijany od 2005 roku. Istnieje wprawdzie jego następca wxDev-C++, ale nie jest jeszcze tak powszechny. Jednak miłośnik wolnego oprogramowania pokazują, że nie poddają się. Od 2011 roku holenderski programista o ksywie Orwell (Johan Mes) kontynuuje rozwój programu.

4. Dobrze jest też mieć kogoś znajomego, kto w razie czego posłuży pomocą. Jeśli nie mamy nikogo takiego, warto skorzystać z Internetu. Na różnych forach i stronach znajdziemy mnóstwo podpowiedzi, materiałów, pomocy i rozwiązań. Trzeba tylko trochę pogonić Google do pracy.

A więc bogaci o te informacje uruchomimy edytor. W niniejszej pracy będę korzystał z edytora Dev-C++, ale większość z nich wygląda i zachowuje się podobnie.

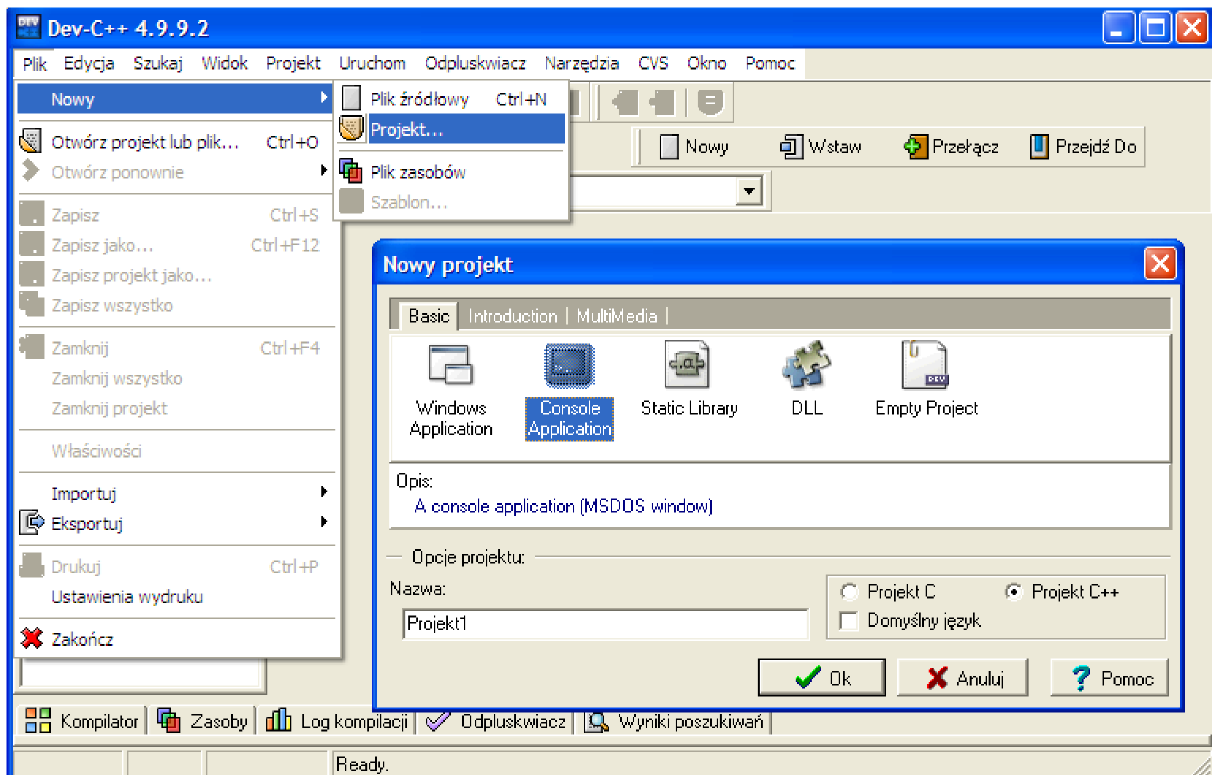
## *Edytor Dev-C++ dla początkujących*

Jeśli go nie posiadamy, trzeba go poszukać w Internecie i ściągnąć. Cały proces nie jest trudny, więc go pominę i przejdę od razu do pracy z C++.

W edytorze Dev-C++ pisany program składa się z projektu, który scala pliki z kodem programu w jedną całość. Plik projektu ma rozszerzenie `.dev`, a pliki z kodem programu `.cpp`. Tworzone są także pliki zasobów (`.o`) oraz wykonawczy `.exe` po kompilacji.

Po uruchomieniu programu pojawia się puste okno Dev-C++. Chcąc uruchomić nowy projekt, wybieramy z menu programu PLIK, potem NOWY i PROJEKT.

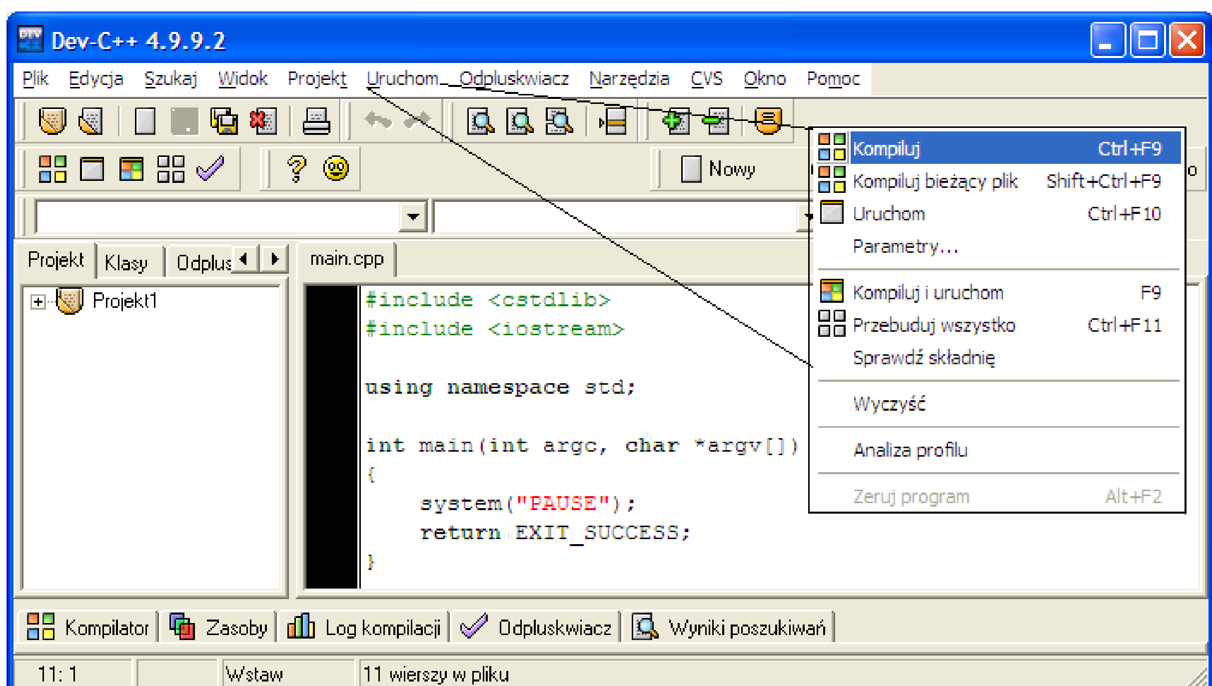
Pojawia się okienko NOWY PROJEKT. Mamy kilka typów do wyboru. Najprostsze jest CONSOLE APLICATION, czyli program uruchamiający się w konsoli tekstowej (takie jakie się tworzyło w MS-DOSie). Brak wprawdzie graficznych ozdobników, ale za to kod programu jest bardzo prosty, co dla początkujących programistów jest bardzo ważne.




**Rysunek 1 - Uruchomienie nowego projektu w Dev-C++**

Potem pojawia się okienko zachęcające do zapisu projektu. Tutaj pewna cenna uwaga: należy utworzyć oddzielny folder dla danego projektu. Domyślnie pliki mają nazwy `projekt1` i `main`. Jeśli będzie ich więcej w danym katalogu, możemy sobie nadpisać efekty wcześniejszej pracy. Odpowiednio nazywając katalogi utrzymamy porządek w naszych projektach.

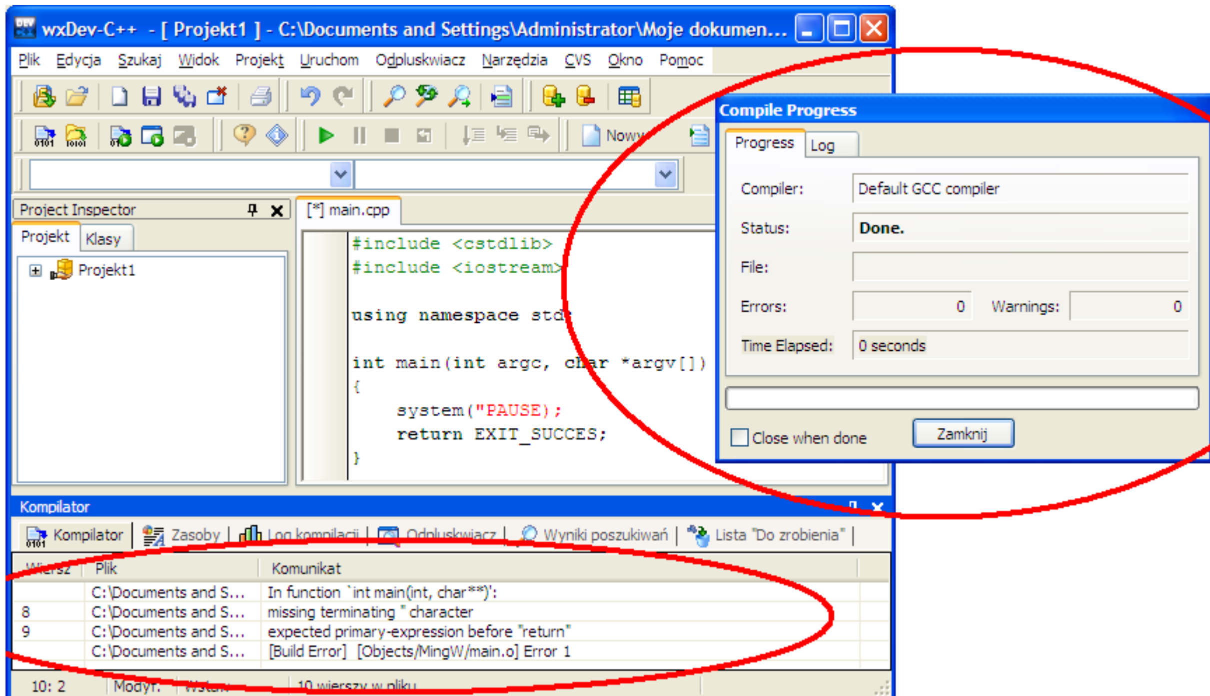
Po zakończeniu zapisywania widzimy utworzony szablon programu.



Gotowy program należy uruchomić. W tym celu wybieramy z menu URUCHOM i odpowiednią pozycję z listy. Każdy program należy najpierw skompilować a potem uruchomić.

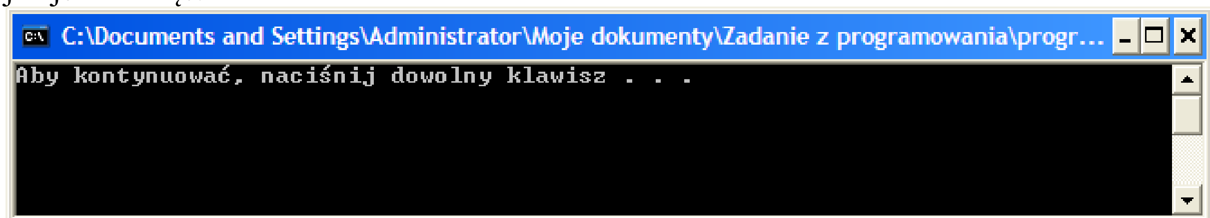
Kompilacja to sprawdzenie poprawności działania programu. Jeśli jest poprawny, to pojawia się stosowne okienko, a jeśli gdzieś zrobiliśmy błąd pojawi nam się lista z informacją o rodzaju błędu i miejscu jego popełnienia.

Czym się różni kompilacja od uruchomienia?




**Rysunek 2 - Lista błędów (numery linii 8 i 9) i okienko poprawnej kompilacji (komunikat *Done*)**

Efekt nie jest zbyt imponujący. Pojawia się czarne okienko z napisem informującym jak je zamknąć.




**Rysunek 3 - Okienko z działającym programem**

Spróbujmy więc napisać coś własnego. Na początek niech to będzie nieśmiertelny przykład „`Hello World!`".

## *Pierwszy program w C/C++*

Wprowadźmy do edytora następujący program :

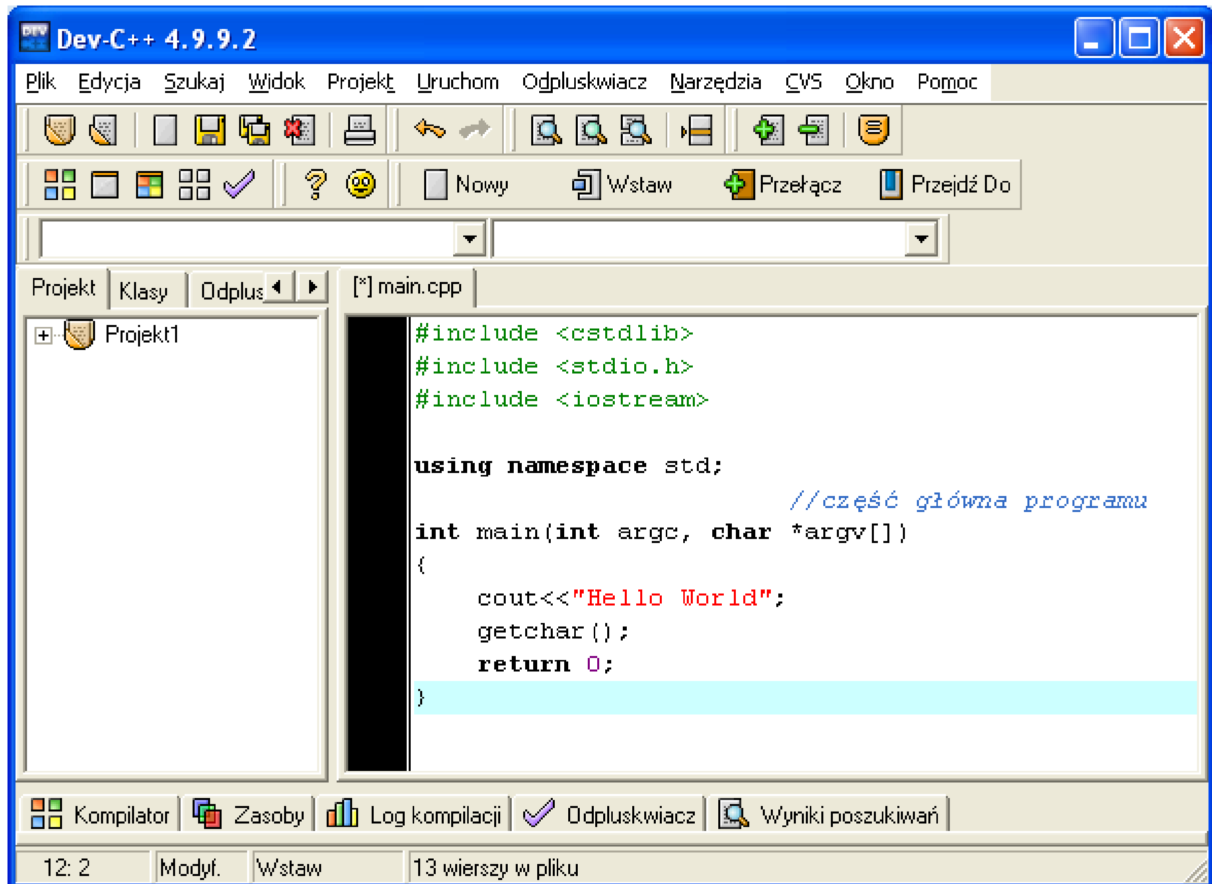
```
#include <cstdlib>
#include <stdio.h>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    cout<<"Hello World";
    getchar();
    return 0;
}
```

**Program 1 – Hello World**

Zauważamy kilka ciekawych rzeczy w edytorze jeśli chodzi o wygląd programu.




**Rysunek 4 - Widok programu w edytorze Dev-C++**

Przede wszystkim niektóre fragmenty są zapisywane innym kolorem lub zaznaczane w inny sposób. To ogromna wygoda dla programistów, gdyż na pierwszy rzut oka[1] można rozpoznać charakterystyczne struktury programu:

- Kolorem zielonym zaznaczane są *biblioteki*,
- Czerwonym *tekst* wypisywane na ekranie,
- Niebieskim *komentarze*
- Wytłuszczone są *kwalifikatory* języka C++, takie jak *typy danych* i *słowa zastrzeżone*.

Ten prosty przykład pozwala zorientować się jak wygląda budowa programu w C++.

## *Budowa programu w C/C++*

### Biblioteki

Pierwszą częścią są *biblioteki*. Są to napisane już przez kogoś często używane polecenia, operacje i algorytmy. Pozwalają nam na ich ponowne użycie, co powoduje, że możemy znacznie uprościć program, wykorzystując struktury stworzone dla naszej wygody.

Biblioteki języka C++

Pierwsza z nich `cstdlib`, ma pełna nazwę ***C Standard** General Utilities **Lib**rary*. A więc jest to standardowa biblioteka języka C i zawiera podstawowe polecenia przydatne w codziennej pracy z C/C++.

Druga z nich `stdio.h` to ***C Standard** **I**nput and **O**utput Library*. Służy do komunikacji programu z otoczeniem. Zawiera operacje potrzebne do wczytywania danych z zewnątrz i wypisywania komunikatów. My ją potrzebujemy do obsługi polecenia `getchar()`.

Trzecia `iostream` to **I**nput/**O**utput **Stream**. Służy (jak wcześniejsza) do wczytywania danych z zewnątrz i wypisywania komunikatów. Tu jest potrzebna do obsługi polecenia `cout<<`.

### Przestrzeń nazw

Kolejną częścią jest *przestrzeń nazw*. To polecenie `using namespace std`. Informuje ono kompilator, że program używa standardowej (`std`) przestrzeni nazw.

Przestrzeń nazw

Do czego się używa przestrzeni nazw? Przestrzeń nazw to zbiór poleceń, nazw, funkcji, które są używane do programowania. Każda przestrzeń to oddzielny zbiór (niezależny od innego).

Po co je stosujemy? Otóż przestrzenie nazw zapewniają komfort programiście. Pisząc jakąś funkcję, nie musi się martwić jak się ją nazwie, ponieważ nazwa nie będzie się gryzła z nazwami innych funkcji w odmiennych przestrzeniach nazw. Co więcej w obrębie różnych przestrzeni nazw, można tworzyć funkcje o takich samych nazwach i deklaracjach, ale robiące inne operacje.

### Funkcja główna

Jest to linijka `int main(int argc, char *argv[])`. Od niej zaczyna się zasadnicza część programu.

Główna część programu

Co znaczy ta linijka i te wszystkie składniki?

[1] Pamiętajmy, że ludzie (a więc i programiści) są wzrokowcami.

- Słowo `main` oznacza, że to jest funkcja główna.
- Elementy zawarte w nawiasie to parametry z jakimi ma się uruchamiać ta funkcja. `argc` oznacza liczbę parametrów, a `argv[]` to tablica wskaźników.
- `int` zawarte na początku linii, to wartość zwracana przez tą funkcję.

Jak na początek wydaje się to strasznie skomplikowane i trudne. Proponuję uznać, że tak ma być i nie myśleć za dużo nad tym na początku nauki, by się nie zniechęcić.

## Wykonywane polecenia

Wykonywane polecenia

W funkcji głównej są umieszczone polecenia, które mają być wykonane. Jak widać do grupowania poleceń służą nam klamry { }[2].

Pierwsze z tych poleceń `cout<<"Hello World";` służy do wypisania tekstu na ekranie. Po poleceniu `cout` umieszcza się dwa znaki większości[3] <<, a po nich tekst w cudzysłowie "", który ma być wypisany.

Druga instrukcja to `getchar();`, które tak naprawdę nie robi nic. Ma ono służyć do zatrzymania programu, by można było zobaczyć efekty działania programu.

Ostatnia linijka to `return 0;`. Zwraca ona 0 (zero). W C++ każda użyta funkcja powinna zwracać jakąś wartość. Ponieważ funkcja `main` jest typu `int` (liczba całkowita) zwraca np. zero. Oczywiście można wpisać inną liczbę, ale powszechnie przyjmuje się wpisywanie na końcu zera.

## *Polecenie cout*

Na pewno zauważyliśmy, że poleceniem, które wypisuje tekst na ekranie jest cout[4].

---

[2] Przez programistów są one ładnie nazywane nawiasami syntaktycznymi.
[3] Zwane krokodylkami.
[4] Czytamy je jako Si-aut.

## *Zmienne w języku C/C++*

Zmienne w języku programowania reprezentuje

tylko 2 operatory tego typu. Ich użycie zademonstruje nam poniższy program.

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

int main(int argc, char *argv[])
{
  int a,b,c;        //deklaracja zmiennych całkowitych
  float x,y,x;      //deklaracja zmiennych zmiennoprzecinkowych
  boolean tak;      //deklaracja zmiennej logicznej

  c= -5;
  cout<<c<<endl;
  c= +5;
  cout<<c<<endl;
  c= 5;
  cout<<c<<endl;


  getch();
  return 0;
     }
```

**Program 2 - Przykład użycia operatorów zmiany znaku**

Zauważymy, że efektem jeg

## *Stałe w języku C/C++*

Stała w języku programowania reprezentuje konkretną, niezmienną wartość numeryczną lub tekstową. Stała podobnie jak zmienna może być całkowita, zmiennoprzecinkowa, wyliczeniowa, logiczna, znakowa.

## *Nazwy w języku C/C++*

Nazwy muszą spełniać wiele zastrzeżeń.

### Słowa kluczowe

Nazwa nie może być taka sama jak słowa kluczowe w C++. Ich lista jest umieszczona poniżej. Została opracowana na podstawie normy ISO/IEC 9899-1999 (C99). Słowa nie zawarte w niej są zaznaczone kursywą.

**Tabela 1 -Słowa kluczowe w języku C++**

| | | | | | |
|---|---|---|---|---|---|
| *asm* | default | goto | register | *template* | volatile |
| auto | do | if | restrict | *this* | while |
| break | double | inline | return | *throw* | _Bool |
| case | else | int | short | *try* | _Complex |
| *catch* | enum | long | signed | typedef | _Imaginary |
| char | extern | *new* | sizeof | union | |
| *class* | float | *operator* | static | unsigned | |
| const | for | *private* | struct | virtual | |
| continue | *friend* | *protected* | switch | void | |

Dlaczego nie wolno tych słów użyć jako nazw zmiennych?

To proste. Kompilator napotykając takie polecenie nie widziałby, czy chodzi o nazwę zmiennej, czy o polecenie. Błędny wybór doprowadziłby do niewłaściwego działania programu. Dlatego musimy tworzyć jednoznaczny kod programu.

Dlaczego nie wolno jako nazw zmiennych używać słów kluczowych?

### Notacja węgierska

Nazwa.

# Notacja węgierska

Z Wikipedii, wolnej encyklopedii

Brak wersji przejrzanej

Skocz do: nawigacji, szukaj

**Notacja węgierska** (ang. *Hungarian Notation*) – w programowaniu sposób zapisu nazw zmiennych oraz obiektów, polegający na poprzedzaniu właściwej nazwy małą literą (literami) określającą rodzaj tej zmiennej (obiektu).

Notację węgierską wymyślił Charles Simonyi, programista z Microsoft. Podane niżej przedrostki są tylko jednym z przykładów formatu zapisu zmiennych w notacji węgierskiej. Tak naprawdę sposobów jest tyle, ilu jest programistów korzystających z tej notacji. Różnie także nazywane są zmienne. Przykładowo można spotkać się z formatami:

iLiczba

i_Liczba

i_liczba

Przykład użycia Notacji węgierskiej do nazywania zmiennych w C++:

| **przedrostek** | **znaczenie** |
| --- | --- |
| s | string (łańcuch znaków) |
| sz | string (łańcuch znaków zakończony bajtem zerowym - null'em) |
| c | char (jeden znak), również const - wartość stała (szczególnie w przypadku użycia wskaźników) |
| by | byte, unsigned char |
| n | Short |
| i | Int |
| x, y | int (przy zmiennych określających współrzędne) |
| cx, cy | int (przy zmiennych określających rozmiar, długość) |
| l | Long |
| w | Word |
| dw | Dword |
| b | boolean (wartość logiczna: prawda lub fałsz) |
| f | Flaga |
| fn | Funkcja |
| h | handle (uchwyt) |
| p | pointer (wskaźnik) |

Przykład użycia Notacji węgierskiej do nazywania obiektów w C#:

| **przedrostek** | **znaczenie** |
| --- | --- |
| asm | Assembly |

| | |
|---|---|
| bln | Boolean |
| btn | Button |
| ch | Char |
| cbx | CheckBox |
| cmb | ComboBox |
| ctr | Container |
| dcol | DataColumn |
| dgrid | DataGrid |
| dgdtpc | DataGridDateTimePickerColumn |
| dgts | DataGridTableStyle |
| dgtbc | DataGridTextBoxColumn |
| dreader | DataReader |
| drow | DataRow |
| dset | DataSet |
| dtable | DateTable |
| date | DateTime |
| dialog | Dialog |
| dr | DialogResult |
| dbl | Double |
| ex | Exception |
| gbx | GroupBox |
| htbl | HashTable |
| iml | ImageList |
| int | Integer |

| | |
|---|---|
| lbl | Label |
| lbx | ListBox |
| lv | ListView |
| rmt | MarshallByRedObject |
| mm | Mainmenu |
| mi | MenuItem |
| frame | MDI-Frame |
| sheet | MDI-Sheet |
| nud | NumericUpDown |
| pnl | Panel |
| pbx | PictureBox |
| rbtn | RadioButton |
| form | SDI-Form |
| sqlcom | SqlCommand |
| sqlcomb | SqlCommandBuilder |
| sqlcon | SqlConnection |
| sqlda | SqlDataAdapter |
| stb | StatusBar |
| str | String |
| strb | StringBuilder |
| tabctrl | TabControl |
| tabpage | TabPage |
| tbx | TextBox |
| tbr | ToolBar |

| tbb | ToolBarButton |
|---|---|
| tmr | Timer |
| usr | UserControl |
| wpl | WindowsPrincipal |

Charakterystyczne dla notacji węgierskiej (tworzące jej "węgierskie brzmienie") są również złożenia przedrostków, zbliżone do składania morfemów gramatycznych z morfemami znaczeniowymi w języku węgierskim (i innych językach aglutynacyjnych).

| **przedrostek** | **znaczenie** |
|---|---|
| lpcsz | "długi" ("daleki") wskaźnik na stały ciąg znaków zakończony bajtem zerowym |
| pfn | wskaźnik na funkcję |

Notacja węgierska, przez wielu wręcz uwielbiana, u innych wywołuje mieszane uczucia. Główną wadą tego systemu jest to, że jeśli chce się zmienić typ zmiennej, trzeba poprawiać nazwę w każdym miejscu programu. W związku z tym notacja nie jest najlepszym rozwiązaniem dla programistów nieco roztargnionych (a także programistów języków dynamicznych).

Notację węgierską stosuje się także w celu zaznaczenia zasięgu zmiennej. Przykładowo:

- g_iZmienna : zmienna globalna, integer
- m_iZmienna : zmienna w strukturze lub klasie, integer
- m_Zmienna : zmienna w strukturze lub klasie
- s_Zmienna : zmienna statyczna klasy
- _Zmienna : zmienna lokalna

sss

# Operatory w C/C++

Jak każdy szanujący się język programowania, również C++ posiada symbole umożliwiające zapisywanie różnych ważnych operacji. Nazywane są operatorami.

## *Operatory wieloargumentowe*

Operatory języka C++ dzielimy w zależności od tego ile argumentów jest potrzebnych do działania.

**Tabela 2 - podział operatorów w zależności od ilości argumentów**

<table>
<tr><th>Ilość argumentów</th><th>Operatory</th></tr>
<tr><td>Jednoargumentowe</td><td>+ - (zmiana znaku) ++ -- sizeof(type) & * (wskaźnikowe) ! ~</td></tr>
<tr><td>Dwuargumentowe</td><td>* / % + - < <= > >= == != *= /= %= += -= &= ^= |= <<= >>= = , ( ) [ ] -> . | && || << >></td></tr>
<tr><td>Trzyargumentowe</td><td>?:</td></tr>
</table>

Poniżej zostaną omówione poszczególne operatory.

## *Operatory zmiany znaku*

Najczęściej używane (i zarazem najbardziej znane) są operatory matematyczne. Pozwalają na zapisanie 4 podstawowych działań matematycznych i reszty z dzielenia.

**Tabela 3 - operatory zmiany znaku w C/C++**

| – | Zmiana znaku | `x= -5;` |
|---|---|---|
| + | Tożsamość znaku (bez zmiany) | `x= +5;` |

Mamy tylko 2 operatory tego typu. Ich użycie zademonstruje nam poniższy program.

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

int main(int argc, char *argv[])
{
  int a,b,c;        //deklaracja zmiennych
                   //działanie operatorów
  c= -5;
  cout<<c<<endl;
  c= +5;
  cout<<c<<endl;
  c= 5;
  cout<<c<<endl;


  getch();
  return 0;
      }
```

**Program 3 - Przykład użycia operatorów zmiany znaku**

Zauważymy, że efektem jego działania są dla 2 i 3 przypadku te same liczby: 5. Widzimy więc, że drugi z operatorów zmiany znaku (+) można pominąć, ponieważ gdy w C++ liczba występuje bez znaku, to domyślnie jest dodatnia.

## *Operatory matematyczne*

Najczęściej używane (i zarazem najbardziej znane) są operatory matematyczne. Pozwalają na zapisanie 4 podstawowych działań matematycznych i reszty z dzielenia.

**Tabela 4 - operatory matematyczne w C/C++**

| | | |
|---|---|---|
| + | Dodawanie | `y=x+5;` |
| - | Odejmowanie | `y=x-5;` |
| * | Mnożenie | `y=x*3;` |
| \ | Dzielenie | `y=x\12;` |
| % | Reszta z dzielenia (dzielenie modulo) | `y=15%7;` |
| = | Podstawienie | `x=5;` |

Na podstawie tej tabeli napiszmy program, który demonstruje działanie operatorów.

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

int main(int argc, char *argv[])
{
  int a,b,c;        //deklaracja zmiennych
               //wprowadzenie wartości
  cout<<"Podaj a";
  cin>>a;
  cout<<"Podaj b";
  cin>>b;
               //działanie operatorów
  c = a + b;
  cout<<”dodawanie ”<<c<<"\n";
  c = a - b;
  cout<<”mnożenie  ”<<c<<"\n";
  c = a * b;
  cout<<”mnożenie  ”<<c<<"\n";
  c = a / b;
  cout<<”dzielenie ”<<c<<"\n";
  c = a % b;
  cout<<”modulo    ”<<c<<"\n";

  getch();
  return 0;
}
```

**Program 4 - operatory arytmetyczne**

Twórcy C/C++ pomyśleli też o ułatwieniu sobie życia przy często używanych operacjach. Dotyczy to pięciu podstawowych działań. Ich zestawienie jest w poniższej tabeli.

**Tabela 5 - operatory skróconych działań matematycznych**

| Operator | Nazwa | Przykład | Równoważne |
|---|---|---|---|
| `+=` | Dodaj | `y+=5;` | `y=y+5;` |
| `-=` | Odejmij | `y-=5;` | `y=y-5;` |
| `*=` | Pomnóż przez | `y*=3;` | `y=y*3;` |
| `\=` | Podziel przez | `y\=12;` | `y=y\12;` |
| `%=` | Podziel modulo przez | `y%=7;` | `y=y%7;` |

A teraz jak zwykle przykład.

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

int main(int argc, char *argv[])
{
  int a,b,c;       //deklaracja zmiennych

  a=5;
  a = a + 3;
  cout<<"dodawanie "<<a<<"\n";
  a=5;
  a += 3;
  cout<<"dodawanie skrócone "<<c<<"\n";

  getch();
  return 0;
}
```

**Program 5 - operatory skróconych działań**

W obydwu przypadkach wynik powinien wynosić 8. Inne operatory proszę dopisać samemu.

## *Operatory krementacji*

Krementacja to zmiana zmiennej o 1. Rozróżniamy 2 operatory: zwiększenia o 1 – *inkrementacja* i zmniejszenia 0 1 – *dekrementacja.*

Odpowiadają one operacjom dodawania i odejmowania o 1, co widać poniżej:

```
x++;    x=x+1;
x--;    x=x-1;
```

Operatory zostały zestawione w poniższej tabeli:

**Tabela 6 - operatory krementacji**

| Dekrementacja (zmniejszenie o 1) | | |
|---|---|---|
| `--` | Predekrementacja | `--x;` |
| | Posdekrementacja | `x--;` |
| Inkrementacja (zwiększenie o 1) | | |
| `++` | Preinkrementacja | `++x;` |
| | Postinkrementacja | `x++;` |

Łatwo zauważyć, że każdy z nich występuje w dwóch wariantach. W pierwszym z nich działanie znajduje się przed, a w drugim po zmiennej.

**Jak jest różnica pomiędzy nimi**?
Najlepiej napisać program i samemu się przekonać jak się zachowują.

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

int main(int argc, char *argv[])
{
  int x,y;       //deklaracja zmiennych
               //krementacja
  x=10;
  cout<<x<<endl;
  cout<<"x++ "<<x++<<" "<<x<<endl;

  x=10;
  cout<<x<<endl;
  cout<<"++x "<<++x<<" "<<x<<endl;

  x=10;
  cout<<x<<endl;
  cout<<"x-- "<<x--<<" "<<x<<endl;

  x=10;
  cout<<x<<endl;
  cout<<"-- x "<<--x<<" "<<x<<endl;

    getch();
  return 0;
}
```

**Program 6 - operatory krementacji**

W wyniku otrzymamy:

Czym się różni postkrementacja od prekrementacji?

**x++ 10 11** **x-- 10 9**
**++x 11 11** **-- x 9 9**

Najprościej rzecz biorąc, w operatorach prekrementacji najpierw jest wykonywane działanie (+1 lub -1), a dopiero potem zmienna jest używana. W operatorach postkrementacji najpierw zmienna jest używana, a dopiero potem jest wykonywane działanie (+1 lub -1).

Nie ma to jednak znaczenia, gdy taka instrukcja występuje samodzielnie. Wtedy wszystkie trzy instrukcje krementacji są równoważne i można je stosować zamienne.

```
x=x+1;  x++;  ++x;
x=x-1;  x--;  --x;
```

## *Operatory relacyjne*

Operatory relacyjne to znaki równości i nierówności (z różnymi ich odmianami). Ich zestawienie znajduje się poniżej.

**Tabela 7 - operatory relacyjne**

| > | Większy | x>y |
|---|---|---|
| >= | Większy lub równy | x>=y |

| < | Mniejszy | `z<t` |
|---|---|---|
| <= | Mniejszy lub równy | `z<=t` |
| == | Równy | `x==5` |
| != | Różny od | `x!=7` |

Operatory te zostaną omówione przy funkcji alternatywy IF.

## *Operatory logiczne*

Operatory logiczne pozwalają na połączenie zmiennych relacjami występującymi w logice matematycznej.

**Tabela 8 - operatory logiczne**

| && | Koniunkcja (AND) | `c = a && b;` |
|---|---|---|
| \|\| | Alternatywa (OR) | `c = a \|\| b;` |
| ! | Negacja (NOT) | `c = !a;` |

Operatory te zostaną omówione później.

## *Operatory bitowe*

Operatory bitowe umożliwiają dokonanie operacji bezpośrednio na bitach w bajtach i słowach.

**Tabela 9 - operatory bitowe**

| & &= | Iloczyn bitowy (AND) | `a = b & c` |
|---|---|---|
| \| \|= | Suma bitowa (OR) | `a = b \| c` |
| ^ ^= | Różnica symetryczna (XOR) | `a = b ^ c` |
| ~ ~= | Uzupełnienie do 1 (NOT) | `a = ~b` |
| >> >>= | Przesunięcie w prawo | `a = b >> 1` |
| << <<= | Przesunięcie w lewo | `a = b << 1` |

Operatory te zostaną omówione później.

## *Operator warunkowy*

Operator warunkowy **?:** jest operatorem trójskładnikowym.

**Tabela 10 - operator warunkowy**

| ?: | Operator warunkowy | `x<0 ? x--: x++;` |
|---|---|---|

Operator ten zostanie omówiony później przy instrukcji IF.

## *Operatory wskaźnikowe*

Operatory wskaźnikowe **&** i ***** umożliwiają dostęp i pracę na wskaźnikach.

**Tabela 11 - operatory wskaxnikowe**

| & | Adres miejsca pamięci | `m=&licznik;` |
|---|---|---|
| * | Wartość zmiennej w danym miejscu pamięci | `x=*m;` |

Operatory te zostaną omówione później.

## *Operator rozmiaru*

Operator rozmiaru **sizeof** podaje wielkość zmiennej lub typu w bajtach.

**Tabela 12 - operator rozmiaru**

| | | |
|---|---|---|
| `sizeof` | Operator rozmiaru zmiennej | `cout<<x<<sizeof x;` |

Operator ten zostanie omówiony później.

## *Operator wiązania*

Operator wiązania **,** stosowany jest do wiązania ze sobą większej liczby wyrażeń. Oddziela inicjatory i argumenty funkcji.

**Tabela 13 - operator wiązania**

| | | |
|---|---|---|
| `,` | Operator wiązania | `x=(y=y-5, 25/y);` |

Ten operator należy do oszczędzających pracę programisty. Przykład poniżej.

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

int main(int argc, char *argv[])
{
  int x,y;       //deklaracja zmiennych
               //tradycyjny sposób
  x=12;
  x=x-6;
  y=24/x;
  cout<<"y= "<<y<<endl;
               //skrócony sposób
  x=12;
  y=(x=x-6, 24/x);
  cout<<"y= "<<y<<endl;

    getch();
  return 0;
}
```

**Program 7 - przykład działania operatora wiązania**

## *Operatory wyboru pola*

Operatory wyboru pola **.** i **->** umożliwiają dostęp do poszczególnych elementów struktur i unii.

**Tabela 14 - operatory wyboru pola**

| | | |
|---|---|---|
| `.` | Dostęp do struktury i unii | `x.a=12;` |
| `->` | Dostęp do wskaźnika do struktury i unii | `x1->a=12;` |

Operatory te zostaną omówione później.

## *Operatory nawiasowe*

W C++ występują trzy rodzaje nawiasów: ( ), [ ], { }.

**Tabela 15 - operatory nawiasowe**

| | | |
|---|---|---|
| ( ) | Nawias do podawania argumentów funkcji i ustalania priorytetów wyrażeń logicznych i arytmetycznych | `y = sin(x);`<br>`y = 2*(a+b);` |
| [ ] | Indeks tablicy | `y = a[2];` |
| { } | Nawiasy syntaktyczne. Służą do grupowania operacji. | `{`<br>`  y=x+5;`<br>`  cout<<y;`<br>`}` |

## *Priorytety operatorów*

Zestawienie operatorów zawiera poniższa tabela.

Operatory jednoargumentowe (*****, **&**, **-**, + ) i trójargumentowy **?** łączą się od strony prawej do lewej, pozostałej od lewej do prawej.

**Tabela 16 - Priorytet operatorów**

| Priorytet | Arytmetyczne | Logiczne i bitowe | Relacyjne | Inne |
|---|---|---|---|---|
| *1 (Najwyższy)* | | | | ( ) [ ] -> . |
| 2 | + - (jednoargumentowe) ++ -- | ! ~ | | sizeof (type) & * (wskaźnikowe) |
| *3* | * / % | | | |
| *4* | + - | | | |
| *5* | | << >> | | |
| *6* | | | < <= > >= | |
| *7* | | | == != | |
| *8* | | & | | |
| *9* | | ^ | | |
| *10* | | \| | | |
| *11* | | && | | |
| *12* | | \|\| | | |
| *13* | | ?: | | |
| *14* | *= /= %= += -= | &= ^= \|= <<= >>= | | = |
| *15 (Najniższy)* | | | | , |

Wiem, że to dość trudne i kłopotliwe na pierwszy rzut oka. Jednak w trakcie programowania szybko można dojść do wprawy i połapać się jak działają poszczególne operatory. W razie czego zawsze można zajrzeć znów do tego rozdziału.

# Proste obliczenia matematyczne

Ponieważ już poznaliśmy operatory języka C++ najlepiej je przećwiczyć np. na obliczeniach matematycznych. Świetnie się do tego nadaje większość z nich.

Przy omawianiu operatorów arytmetycznych widzieliśmy już program demonstrujący działanie podstawowych operatorów matematycznych (+-*/%). Zobaczmy teraz jak możemy posłużyć się nimi przy pisaniu bardziej rozbudowanych działań.

## *Zapis wybranych konstrukcji matematycznych*

Jak zapisywać mnożenie w C++?

Równanie **c=2a+3b** zapiszemy jako `c= 2*a + 3*b;`. Konstrukcje typu **2a** musimy wyraźnie zapisywać jako mnożenie `2*a`. Skróconego sposobu możemy używać na co dzień, przy nauce matematyki, ale kompilator niestety nie rozumie tego sposobu. **2a** potraktuje jako zmienną `2a`, a nie mnożenie `2*a`.

Jak zapisywać potęgi?

Potęgi możemy zapisać jako iloczyn tej samej zmiennej. Niestety symbol ^ używany do opisu potęg nie działa w C++.

$c=a^2+b^2$ `c = a*a + a*b;` $c=a^3$ `c = a*a*a;`

Jak ustalać priorytety działań?

Przy różnych priorytetach operatorów używamy nawiasów wymuszających odpowiednią kolejność działań. Jak widać poniżej, opuszczenie ich spowoduje, że będą to dwa zupełnie różne działania.

$c=2(a-b)$ `c=2*(a-b);`

$c=2a-b$ `c=2*a - b;`

Jak zapisać ułamki?

Jak pamiętamy ze szkoły, kreska ułamkowa zastępuje nam dzielenie. W C++ możemy analogicznie ją zmienić na dzielenie.

$c=\frac{a}{b}$ `c= a / b;`

Przy zapisie ułamków szczególnie ważne jest używanie nawiasów, które oddzielają licznik od mianownika i wymuszają jednocześnie priorytety działań. Przy większych działaniach w liczniku czy mianowniku lepiej zawsze dodać jedną parę nawiasów więcej „na wszelki wypadek”.

$c=\frac{a-2}{b+2}$ `c= (a-2)/(b+2);`

$c=a-\frac{2}{b}+2$ `c= a - 2/b + 2;`

Innym często używanym działaniem są potęgi rozbudowanych wyrażeń. Zapisujemy korzystając z wiedzy o nawiasach i zapisie potęg.

$c=(a+b)^2$ `c=(a+b) * (a+b);`

$c=(a-b)^3$ `c=(a-b)*(a-b)*(a-b);`

Bardziej skomplikowane wyrażenia piszemy analogicznie, ale musimy być uważni, by nie popełnić błędu.

$$c = \frac{a^2 - 2b^3 + ab}{2a - a^3 + b^2 - 3b}$$

```
c=(a*a-2*b*b*b+a*b)/(2*a-a*a*a+b*b-3*b);
```

Tyle krótkiego wprowadzenia. Teraz kilka ćwiczeń.

## *Ćwiczenia*

1. Napisz program, który wczytuje dany rok i podaje ile lat temu miała miejsce Bitwa pod Grunwaldem.
2. Napisz program, który wczytuje twój wiek i podaje, w którym roku się urodziłeś.
3. Napisz program liczący:
    a. Pole kwadratu
    b. Pole prostokąta
    c. Pole trójkąta
4. Napisz program, który wczytuje kwotę pieniędzy netto i podaje:
    a. Ile wynosi VAT (23%)
    b. Ile wynosi kwota brutto
5. Wykorzystując operatory zapisz w C++ następujące proste wzory matematyczne:
    a. $z = (x+y)^2$
    b. $z = x^2 + y^2$
    c. $z = x^3 + y^2$
    d. $z = (x+3)^2$
    e. $z = (x-3)^2$
    f. $z = x * (x+y)$
    g. $z = \frac{x^2 + y^2}{x - y}$
    h. $z = \frac{(x+y)^2}{x-y}$
    i. $z = \frac{x+y}{(x-y)^2}$
    j. $z = \frac{x^2 + y^2}{x^2 - y^2}$
    k. $z = \left(\frac{x+3}{x-3}\right)^2$
6. Napisz program, który wczytuje dwie liczby i zamienia ich wartości (*wskazówka*: dodaj trzecią pomocniczą)
7. Napisz program wczytujący liczbę całkowitą i podający ile wynosi liczba set, dziesiątek, jedności.
8. Napisz program wczytujący temperaturę w stopnia Celsjusza i podający ją w stopniach Kelvina (i odwrotnie).
9. Napisz program w czytujący temperaturę w stopnia Celsjusza i podający ją w stopniach Fahrenheita (i odwrotnie).
10. Napisz program, który wczytuje wielkość promienia lub średnicy koła i na jej podstawie wylicza:
    - Pole koła
    - Obwód koła
11. Napisz program, który znając pierwszy wyraz ciągu arytmetycznego, ilość wyrazów i różnice pomiędzy nimi, oblicza wyraz o numerze n.
12. Napisz program, który znając pierwszy wyraz ciągu geometrycznego, ilość wyrazów i różnice pomiędzy nimi, oblicza wyraz o numerze n.

# Funkcje matematyczne

Język C++ posiada możliwość korzystania z bardziej zaawansowanych funkcji matematycznych. Jednym z jego zastosowań jest właśnie dokonywanie obliczeń.

Podstawowe funkcje matematyczne są dostępne po włączeniu standardowej biblioteki `math.h`. W tym rozdziale zostaną omówione te najważniejsze i najczęściej używane[5].

## *Popularne funkcje matematyczne*

Zestawienie często używanych funkcji znajduje się w poniższej tabelce.

**Tabela 17 - Popularne i lubiane funkcje matematyczne języka C++**

| | | |
|---|---|---|
| ceil(x) | Najmniejsza liczba całkowita większa od x | `ceil(3.7);` |
| floor(x) | Największa liczba całkowita mniejsza od x | `floor(3.7);` |
| fmod(x,y) | Reszta z dzielenia x przez y | `fmod(13,5);` |
| hypot(a,b) | Oblicza w trójkącie długość przeciwprostokątnej | `hypot(4,3);` |
| sqrt(x) | Pierwiastek kwadratowy z x | `cout<<sqrt(4);` |
| fabs(x) | Wartość bezwzględna z x | `fabs(-2.7);` |

Najpierw zobaczymy jaki jest efekt użycia poleceń `ceil(x)` i `floor(x)`. Podają liczby całkowite mniejsze i większe od wpisanej. Np. dla **3.7** są to **3** i **4**.

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

using namespace std;

int main(int argc, char *argv[])
{
    float x;

    cout<<"Podaj liczbe rzeczywista";
    cin>>x;
    cout<<endl;
    cout<<"Najmniejsza liczba calkowita wieksza to "<<ceil(x)<<endl;
    cout<<"Najwieksza liczba calkowita mniejsza to"<<floor(x)<<endl;

    cout<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 8 - przykład działania funkcji ceil i floor**

Jako ćwiczenie proponuję sprawdzić reakcję programu na wpisanie liczby całkowitej i ujemnej.

[5] Dostępnych jest wiele bibliotek języka C/C++, które pozwalają na skorzystanie z innych wzorów, równań i procedur matematycznych.

Następne przedstawione niżej to pierwiastek kwadratowy (`sqrt` od *square root*) i wartość bezwzględna (`fabs` od *function absolute*).

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

using namespace std;

int main(int argc, char *argv[])
{
    float x;

    cout<<"Podaj liczbe rzeczywista";
    cin>>x;
    cout<<"Pierwiastek kwadratowy "<<sqrt(x)<<endl;
    cout<<"Wartość bezwzględna "<<fabs(x)<<endl;

    cout<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 9 - przykład działania funkcji sqrt i fabs**

Zostały jeszcze dwie: dzielenie modulo (`fmod` od *function modulo*) i obliczanie przeciwprostokątnej w trójkącie (`hypot` od *hypotenuse*).

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

using namespace std;

int main(int argc, char *argv[])
{
    int a,b,c;

    cout<<"Podaj dzielna ";
    cin>>a;
    cout<<"Podaj dzielnik ";
    cin>>b;
    c=fmod (a,b);
    cout<<"funkcja fmod "<<c<<endl;
    c=a%b;
    cout<<"  operator % "<<c<<endl;

    cout<<"Przeciwprostokatna wynosi "<<hypot(a,b)<<endl;

    cout<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 10 - przykład działania funkcji fmod i hypot**

Funkcja `fmod` dzieli też liczby rzeczywiste (a więc umie więcej niż operator %).

## *Funkcje trygonometryczne*

Zestawienie dostępnych funkcji trygonometrycznych znajduje się w poniższej tabelce.

**Tabela 18 - Funkcje trygonometryczne biblioteki math.h**

| | | |
|---|---|---|
| sin(x) | sinus (x) | `z = sin(1.0);` |
| cos(x) | cosinus (x) | `z = cos(1.0);` |
| tan(x) | tangens (x) | `z = tan(1.0);` |
| asin(x) | arcus sinus (x) | `z = asin(-0.5);` |
| acos(x) | arcus cosinus (x) | `z = acos(-0.5);` |
| atan(x) | arcus tangens(x) | `z = atan(-0.5);` |
| atan2(x,y) | arcus tangens(x/y) | `z = atan2(2,3);` |
| sinh(x) | sinus hiperboliczny (x) | `sh = sinh (0.7);` |
| cosh(x) | cosinus hiperboliczny (x) | `ch = cosh (x);` |
| tanh(x) | tangens hiperboliczny (x) | `th = tanh (-0.1);` |

Poniżej jest przykład korzystania z tych funkcji w programach języka C++

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

using namespace std;

int main(int argc, char *argv[])
{
    float a,b,c;

    cout<<"Podaj liczbe ";
    cin>>a;
    c=sin (a);
    cout<<"  sinus "<<c<<endl;
    c=cos (a);
    cout<<"cosinus "<<c<<endl;
    c=tan (a);
    cout<<"tangens "<<c<<endl;
     c=asin (a);
    cout<<"arcus sinus "<<c<<endl;
    c=acos (a);
    cout<<"arcus cosinus "<<c<<endl;
    c=atan (a);
    cout<<"arcus tangens "<<c<<endl;
    c=sinh (a);
    cout<<"  sinus hiperboliczny "<<c<<endl;
    c=cosh (a);
    cout<<"cosinus hiperboliczny "<<c<<endl;
    c=tanh (a);
    cout<<"tangens hiperboliczny "<<c<<endl;

    cout<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 11 - przykład działania funkcji trygonometrycznych**

Inne funkcje możemy obliczyć korzystając ze znanych nam wzorów trygonometrycznych.

secans $$\sec = \frac{1}{\cos inus}$$

cosecans $$\cos ecans = \frac{1}{\sin us}$$

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

using namespace std;

int main(int argc, char *argv[])
{
    float a,sec,cosec;

    cout<<"Podaj liczbe ";
    cin>>a;

    sec=1/cos (a);
    cout<<"secans "<<sec<<endl;
    cosec=1/sin (a);
    cout<<"cosecans "<<cosecc<<endl;


    cout<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 12 – Obliczanie niestandardowych funkcji trygonometrycznych**

Argumenty funkcji trygonometrycznych są radianami, nie kątami. To może powodować wrażenie błędu. Nic nie stoi na przeszkodzie, żeby dopisać przeliczanie kątów na radiany. Wzór ma postać $rad = \frac{kat \cdot \Pi}{180}$

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

using namespace std;

int main(int argc, char *argv[])
{
    float radian,kat,c;
   const double PI=3.14159265358979323846;

    cout<<"Podaj kat ";
    cin>>kat;
    radian=kat*PI/180;
```

```
    c=sin(radian);
    cout<<endl<<"  sinus "<<c<<endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 13 - przykład przeliczania kątów na radiany**

W ramach ćwiczeń proszę się zastanowić nad przeliczanie radiana na stopnie.

## *Funkcje potęgowe i logarytmiczne*

Zestawienie tych funkcji jest zawarte w poniższej tabelce.

| | | |
|---|---|---|
| log(x) | logarytm naturalny x | `cout<<log(2.71);` |
| log10(x) | logarytm dziesiętny x | `cout<<log10(100);` |
| exp(x) | Eksponenta x ($e^x$) | `cout<<exp(1.0);` |
| pow(a,b); | Oblicza $a^b$ | `z = pow(10,2);` |
| pow10(b); | Oblicza $10^b$ | `z = pow10(2);` |
| ldexp(m,w) | Oblicza wynik $m*2^w$ | `x=ldexp(mant,wykl);` |

Przykłady użycia tych funkcji znajdziemy w poniższych programach.

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

using namespace std;

int main(int argc, char *argv[])
{
    float x,y,z;

    cout<<"Podaj liczbe";
    cin>>x;

    z=log(x);
    cout<<endl<<"logarytm naturalny "<<z<<endl;
    z=log10(x);
    cout<<endl<<"logarytm dziesietny "<<z<<endl;
    z=exp(x);
    cout<<endl<<"eksponenta "<<z<<endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 14 - przykład działania funkcji logarytmicznych**

```
#include <cstdlib>
#include <iostream>
```

```
#include <math.h>

using namespace std;

int main(int argc, char *argv[])
{
    float x,y,z;

    cout<<"Podaj liczbe";
    cin>>x;

    z=log(x);
    cout<<endl<<"logarytm naturalny "<<z<<endl;
    z=log10(x);
    cout<<endl<<"logarytm dziesietny "<<z<<endl;
    z=exp(x);
    cout<<endl<<"eksponenta "<<z<<endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 15 - przykład działania funkcji potęgowych**

Tyle teorii. Czas na ćwiczenia.

## *Ćwiczenia – funkcje trygonometryczne*

1. Zapisz następujące wzory trygonometryczne:

a) $z = \sin(x + y)$

b) $z = \sin(x^2 - 16)$

c) $z = \cos(x + y)$

d) $z = \cos(x - y)$

e) $z = \cos(x^2 + y)$

f) $z = \tan(x + y^2)$

g) $z = \tan(2x - y^2)$

h) $z = \arcsin(\frac{x}{y})$

i) $z = \arccos(\frac{x+y}{x})$

j) $z = \arctan(x - 5)$

k) $z = \arctan(3 - y)$

l) $z = \sinh(x + y^2)$

m) $z = \frac{\sin(x^2) + 3}{\cos(x^2) - 3}$

2. Napisz program sprawdzający czy funkcja tangens da takie same wyniki jak iloraz sinus i cosinus $\tan gens_x = \frac{\sin x}{\cos x}$.
3. Napisz program liczący wartość funkcji cotangens. Porównaj dwa wzory: $\cot angens_x = \frac{\cos x}{\sin x}$ i $\cot angens_x = \frac{1}{\tan x}$.
4. Sprawdź poprawność wzoru na jedynkę trygonometryczną. Wykorzystaj którąś z poniższych tożsamości:
   a. $jed_tryg1 = \sin^2 x + \cos^2 x$
   b. $jed_tryg2 = \sec^2 x - tg^2 x$
   c. $jed_tryg3 = \cos ec^2 x - ctg^2 x$
5. Sprawdź czy poniższe wzory jednego kąta dają takie same wyniki.
   a. $\sin us_x = \sqrt{1 - \cos^2 x} = \frac{tgx}{\sqrt{1 + tg^2 x}}$

b. $\cos inus_x = \sqrt{1-\sin^2 x} = \dfrac{1}{\sqrt{1+tg^2 x}}$

c. $\tan gens_x == \dfrac{\sin x}{\sqrt{1-\sin^2 x}} = \dfrac{\sqrt{1-\cos^2 x}}{\cos x}$

d. $\cot angens_x == \dfrac{\sqrt{1-\sin^2 x}}{\sin x} = \dfrac{\cos x}{\sqrt{1-\cos^2 x}} = \dfrac{1}{tgx}$

6. Sprawdź czy poniższe wzory sumy i różnicy kątów dają takie same wyniki.
   a. $\sin us(x \pm y) = \sin x \cos y \pm \cos x \sin y$
   b. $\cos(x \pm y) = \cos x \cos y \mp \sin x \sin y$
   c. $tg(x \pm y) = \dfrac{tgx \pm tgy}{1 \mp tgxtgy}$
7. Sprawdź czy poniższe wzory na kąty wielokrotne dają takie same wyniki.
   a. $\sin 2x = 2\sin x \cos x$
   b. $\cos 2x = \cos^2 x - \sin^2 x = 2\cos^2 x - 1 = 1 - 2\sin^2 x$
   c. $tg 2x = \dfrac{2tgx}{1-tg^2 x}$
8. Sprawdź czy poniższe wzory na potęgi funkcji dają takie same wyniki.
   a. $\sin^2 x = \dfrac{1-\cos 2x}{2}$
   b. $\cos^2 x = \dfrac{1+\cos 2x}{2}$

## *Ćwiczenia – funkcje logarytmiczne*

1. Zapisz następujące wzory matematyczne:
   a) $z = \ln(x+y)$
   b) $z = \log_{10}(x-y)$
   c) $z = \ln(\frac{x}{y})$
   d) $z = \log_{10}(x-y)$
   e) $z = \log_{10}(x-5)$
   f) $z = \ln(3-y)$
   g) $z = e^{x+y}$
   h) $z = e^{y^2+5}$
   i) $z = e^{\frac{x-y}{x+y}}$
   j) $z = \ln\left|\dfrac{x^2-36}{x-6}\right|$
   k) $z = \left|\dfrac{\log_{10}(x^2-81)}{x-9}\right|$
2. Napisz program liczący wartość logarytmu dla dowolnej podstawy a. Wykorzystaj wzór $\dfrac{\log_b x}{\log_b a} = \log_a x$.

## *Ćwiczenia różne*

1. Zapisz następujące wzory i porównaj ich sposób zapisu w C++:
   a) $z = \sqrt{y+5}$
   b) $z = \sqrt{y} + 5$
   c) $z = y + \sqrt{5}$
   d) $z = \sqrt{y} + \sqrt{5}$

2. Napisz program, który wczytuje współrzędne punktu w układzie współrzędnych i podaje, jaka jest odległość od początku układu współrzędnych.
3. Napisz program, który wczytuje współrzędne dwóch punktów w układzie współrzędnych i podaje, jaka jest odległość pomiędzy nimi.
4. Napisz program, który wczytuje współrzędne trzech punktów w układzie współrzędnych i podaje, jakie jest pole trójkąta pomiędzy nimi.
5. Napisz program, który wczytuje współrzędne trzech punktów w układzie współrzędnych i podaje, jakie współrzędne ma środek ciężkości tego trójkąta (barycentrum).
6. Dane są przyprostokątne trójkąta prostokątnego **a** i **b**. Oblicz przeciwprostokątną **c** oraz kąty trójkąta w stopniach. (Użyj zarówno funkcji `hypot` jak i oblicz ze wzoru pitagorasa).
7. Napisz program obliczający podstawowe parametry koła.
    a. Wczytuje promień i kąt wycinka kołowego.
    b. Podaje: średnicę koła, obwód koła, pole koła, pole wycinka i długość łuku.
8. Napisz program obliczający podstawowe parametry kwadratu.
    a. Wczytuje długość boku.
    b. Podaje: przekątną kwadratu, obwód kwadratu, pole kwadratu, pole koła opisanego na kwadracie i pole koła wpisanego w ten kwadrat.
9. Napisz program obliczający podstawowe parametry prostokąta.
    a. Wczytuje długość boków.
    b. Podaje: przekątną prostokąta, obwód prostokąta, pole prostokąta, pole koła opisanego na prostokąta.
10. Napisz program obliczający podstawowe parametry pięciokąta foremnego.
    a. Wczytuje długość boku.
    b. Podaje: przekątną pięciokąta foremnego, wysokość pięciokąta foremnego, obwód pięciokąta foremnego, pole pięciokąta foremnego, pole koła opisanego na pięciokącie foremnym i pole koła wpisanego w pięciokąt foremny.
11. Napisz program obliczający podstawowe parametry sześciokąta foremnego.
    a. Wczytuje długość boku.
    b. Podaje: obwód sześciokąta foremnego, pole sześciokąta foremnego, pole koła opisanego na sześciokącie foremnym i pole koła wpisanego w sześciokąt foremny.
12. Napisz program obliczający podstawowe parametry ośmiokąta foremnego.
    a. Wczytuje długość boku.
    b. Podaje: obwód ośmiokąta foremnego, pole ośmiokąta foremnego, pole koła opisanego na ośmiokącie foremnym i pole koła wpisanego w ośmiokąt foremny.
13. Napisz program obliczający podstawowe parametry trójkąta równobocznego.
    a. Wczytuje długość boku.
    b. Podaje: wysokość trójkąta równobocznego, obwód trójkąta równobocznego, pole trójkąta równobocznego, pole koła opisanego na trójkącie równobocznym i pole koła wpisanego w trójkąt równoboczny.
14. Napisz program obliczający podstawowe parametry walca.
    a. Wczytuje promień i wysokość walca.
    b. Podaje: objętość walca, pole powierzchni walca.
15. Napisz program obliczający podstawowe parametry kuli.
    a. Wczytuje promień kuli.
    b. Podaje: średnicę, objętość kuli, pole powierzchni kuli.
16. Napisz program obliczający podstawowe parametry stożka obrotowego.
    a. Wczytuje promień podstawy i wysokość.

    b. Podaje: średnicę podstawy, długość tworzącej, objętość stożka, pole powierzchni bocznej stożka, kąt rozwarcia stożka, objętość kuli opisanej na stożku.
17. Napisz program obliczający podstawowe parametry sześcianu.
    a. Wczytuje długość boku.
    b. Podaje: przekątną sześcianu, objętość sześcianu, pole powierzchni bocznej sześcianu, kąt rozwarcia stożka, objętość kuli opisanej na sześcianie.
18. Napisz program obliczający podstawowe parametry czworościanu foremnego.
    a. Wczytuje długość boku.
    b. Podaje: wysokość czworościanu foremnego, objętość czworościanu foremnego, pole powierzchni bocznej czworościanu foremnego, objętość kuli opisanej na sześcianie.
19. S

# Instrukcja IF

Instrukcja IF to instrukcja, która odpowiada nam na pytanie, czy dany warunek jest lub nie jest spełniony.

Instrukcja IF ma postać:

```
if (warunek)
{
    instrukcje_na_tak;
}
else
{
    instrukcje_na_nie;
};
```

Warunek musi być tak skonstruowany, by odpowiedź brzmiała TAK lub NIE. Przykładem takich warunków są: Czy 2+2=4? Czy x>0? Czy reszta z dzielenie wynosi 1?

Taka instrukcja nazywa się alternatywą[6].

Zobaczmy jak wygląda ta instrukcja w programie.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int x,y;
    cout<<"Podaj dwie liczby"<<endl;
    cin>>x;
    cin>>y;

    if (y==x)                    //instrukcja IF
      {cout<<"Liczby sa rowne"<<endl;}
    else
      {cout<<"Liczby nie sa rowne"<<endl;};

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 16 – Przykład instrukcji IF**

Jak widać nie jest to trudne.

## *Warunki instrukcji IF*

Najistotniejsze przy korzystaniu z instrukcji IF jest umiejętne ułożenie warunku. Później idzie już z górki.

Wykorzystamy tu operatory relacyjne. Jest ich sześć.

---

[6] Popularne skojarzenie z serialem *Alternatywy 4* może być mylące. Alternatywa ma tylko 2 wyjścia, a nie 4.

**Tabela 19 - operatory relacyjne**

| > | Większy | x>y |
|---|---|---|
| >= | Większy lub równy | x>=y |
| < | Mniejszy | z<t |
| <= | Mniejszy lub równy | z<=t |
| == | Równy | x==5 |
| != | Różny od | x!=7 |

Możemy wykorzystać dowolne z nich, pamiętając jednak, że trzeba odpowiednio dobrać polecenia jakie ma wykonać instrukcja IF. Niektóre z nich możemy zamienić ze sobą, pamiętając jednak o konieczności korekty wykonywanych poleceń.

Jak to zrobić? Zachęcam do analizy przykładu. Obydwa polecenia mają ten sam efekt, choć warunki są wobec sobie komplementarne (uzupełniające się).

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int x,y;
    cin>>x;

    if (x==10)              //równość  2 liczb
     { cout<<"x to dziesiec"<<endl;}
    else
     { cout<<"x to nie jest dziesiec"<<endl;};

    if (x!=10)              //nierówność  2 liczb
     { cout<<"x to nie jest dziesiec"<<endl;}
    else
     { cout<<"x to dziesiec"<<endl;};


    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 17 – Zastosowanie operatorów relacyjnych**

Zestawiając operatory razem wyróżnimy 3 uzupełniające się pary.

**Tabela 20 - komplementarne pary operatorów relacyjnych**

| >= | < |
|---|---|
| <= | > |
| == | != |

Każdy z operatorów z jednej kolumny może być zastąpiony tym z sąsiedniej. Pamiętać należy jednak, by poprawić wykonywane operacje dla danej odpowiedzi.

## *Operator warunkowy*

Przydatnym ułatwieniem jest operator warunkowy **?:**. W sytuacji gdy mamy mało rozbudowaną instrukcję alternatywy, nie ma potrzeby pisać pełnej instrukcji IF. Do tego celu wymyślono specjalny operator, który jest operatorem trójargumentowym. Należy jednak pamiętać, że wszystkie argumenty muszą być pojedynczymi wyrażeniami.

Ma on postać:

```
warunek ? instrukcja_na_tak : instrukcja_na_nie;
```

Porównajmy jak wyglądają obydwa sposoby zapisu tej samej alternatywy.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int x,y ;
    cin>>y;

    x=y;
    if (y>0)                //klasyczna instrukcja IF
     {x--;}
    else
     {x++;};
    cout<<x<<endl;

    x=y;
    y>0 ? x--: x++;         //operator ?:
    cout<<x<<endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 18 - Porównanie instrukcji IF i operatora ?:**

Dla prostych i niedużych alternatyw, operator **?:** jest wygodniejszy i czytelniejszy.

## *Zagnieżdżone instrukcje IF*

Jeśli mamy takie życzenie możemy stworzyć rozbudowane drzewo warunków. Przy realizacji największym problemem jest ustalenie do którego z IF należy dany ELSE. To jest kłopotliwe zwłaszcza, gdy mamy dużo IFów.

Pierwszym rozwiązaniem jest sumienne stosowanie wcięć oraz klamer, co pozwoli opanować ewentualny chaos.

Drugim jest wykorzystanie właściwości języka C++, który przydziela ELSE najbliższemu, poprzedzającemu je IF, który nie jest związany z innym ELSE. Jeśli chcemy użyć innego powiązania, musimy zastosować klamry.

```
if (x>0)
  if (y>0)
     cout<<"zero";
else
```

```
if (x>0)
{
  if (y>0)
     cout<<"zero";
```

| cout<<”nie zero”; | }<br>else<br>    cout<<”nie zero”; |
|---|---|
| W tym przypadku brak jest klamer, więc ELSE łączy się z `if  (y>0)`, gdyż jest najbliższym, wcześniejszym IF. | W tym przypadku użycie klamer powoduje, że ELSE łączy się z `if  (x>0)`. `If(y>0)`, jest na innym (niższym) poziomie, więc nie jest brane pod uwagę. |

Rozsądnie zrealizowane drzewo jest czytelne i przejrzyste.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int x,y ;
    cin>>x;
    cin>>y;

    if (x!=y)                  //najwyższy poziom IF
    {
     if (y>x)                  //poziom IF  -1
        cout<<"y jest wieksze\n";
     else
        cout<<"x jest wieksze\n";
    }
    else
    {
     cout<<"liczby sa rowne\n";
     if (y>0)                  //poziom IF  -1
        cout<<"sa wieksze od zera\n";
     else
     {
        if (y<0)               //poziom IF  -2
           cout<<" sa mniejsze od zera\n";
        else
           cout<<" sa rowne zero\n";
     }
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 19 – Przykład zagnieżdżonych  instrukcji IF**

Czasem zamiast zagnieżdżonych instrukcji IF lepiej użyć operatorów logicznych.

## *Operatory logiczne*

Operatory logiczne pozwalają na stworzenie warunku zawierającego więcej niż jedną relację. Np. chcemy by program dla x>0 i jednocześnie x<5 realizował inną operację niż dla x

dodatniego, ale większego od 5. Albo szukamy rozwiązania dla x<-1 i x>1. Wtedy musimy użyć któregoś z trzech poniższych operatorów.

**Tabela 21 - operatory logiczne**

| && | Koniunkcja (AND) | `c = a && b;` |
|---|---|---|
| \|\| | Alternatywa (OR) | `c = a \|\| b;` |
| ! | Negacja (NOT) | `c = !a;` |

Najlepiej ich działanie zobaczyć na przykładzie.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int x;
    cin>>x;

    if (x>0 && x<5)              // przykład użycia operatora AND
    {
      cout<<"x jest zawarte w przedziale 0-10"<<endl;
    };

    if (x>1 || x<-1)              // przykład użycia operatora OR
    {
      cout<<"x nie jest w przedziale -1 do 1"<<endl;
    };

    if (x!=0)              // przykład użycia operatora NOT
    {
      cout<<"x nie jest rowne 0"<<endl;
    };

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 20 - Porównanie instrukcji IF i operatora ?:**

Inne funkcje jak EX-OR, NOR można uzyskać za pomocą nawiasów i kombinacji tych 3 funkcji.

**Tabela 22 – uzyskiwanie innych operatorów logicznych**

| Funkcja logiczna | Przykład |
|---|---|
| NAND | `c = !(a && b);` |
| NOR | `c = !(a \|\| b);` |
| XOR | `c = (a && !b) \|\| (!a && b)` |
| XNOR | `c = !(a && b) \|\| (a && b)` |

Inne funkcje logiczne też da się zapisać za pomocą tych trzech podstawowych.

## *Ćwiczenia*

1. Napisz program, który rozpoznaje znak liczby (większa, mniejsza, równa zero).
2. Napisz program, który wczytuje dwie liczby i podaje, czy są sobie równe czy nie. Jeśli nie, program podaje, która jest większa.
3. Napisz program, który rozpoznaje czy liczba jest parzysta, czy nieparzysta.
4. Napisz program, który rozpoznaje czy liczba jest podzielna przez daną liczbę.
5. Dane są trzy boki a,b,c. Napisz program sprawdzający, czy da się z nich zbudować trójkąt.
6. Dane są trzy boki a,b,c. Napisz program sprawdzający, czy da się z nich zbudować trójkąt prostokątny.
7. Napisz program wczytujący liczby z klawiatury i podający, która z nich jest największa (najmniejsza).
8. Napisz program, który wczytuje wzrost i podaje jego kategorię:
    - $0 < x < 165$ niski wzrost
    - $165 \leq x < 175$ średni wzrost
    - $175 \leq x$ wysoki wzrost
9. Napisz program, który wczytuje wiek i podaje jego kategorię:
    - $0 < x < 18$ nieletni
    - $18 \leq x < 35$ osoba młoda
    - $35 \leq x < 65$ osoba w średnim wieku
    - $65 \leq x$ osoba starsza

# Instrukcja SWITCH..CASE

Instrukcja IF-ELSE jest bardzo wygodnym narzędziem. Jednak nie zdaje egzaminu w sytuacjach, gdy musimy wybierać pomiędzy więcej niż 2 sytuacjami. Wyjściem jest zagnieżdżanie instrukcji IF-ELSE wewnątrz siebie, ale komplikuje to kod programu, co utrudnia (a w niektórych przypadkach wręcz uniemożliwia) analizę jego działania.

Do takich przypadków wymyślono funkcję SWITCH..CASE zwaną wielowyborem. Użycie instrukcji wygląda następująco:

```
  switch (wyrażenie)
  {
   case  wynik1:
           instrukcje_przypadku_1
         break;
   case  wynik2:
           instrukcje_przypadku_2
         break;
   …
   default:
        instrukcje_dla_pozostałych_przypadków
  }
```

Działa ona w ten sposób, że porównuje wartość wyrażenia z kolejnymi elementami listy zawierającej pożądane wyniki. Wartości na liście muszą być unikalne. Po znalezieniu właściwej, wykonywane są odpowiednie instrukcje. Zakończenie każdego z przypadków sygnalizowane jest instrukcją `break`. Jeśli nie ma `break`, wykonywane są również polecenia następnego CASE.

Jeśli nie wybrano żadnej wartości znajdującej się na liście, wykonywane są instrukcje znajdujące się po słowie `default` (lub jeśli go nie ma, program nie robi nic).

W odróżnieniu od instrukcji IF, SWITC..CASE może operować tylko na liczbach całkowitych lub stałych znakowych. Sprawdza jedynie zgodność wartości z listy (bez relacji czy wyrażeń logicznych).

Jak działa program wykorzystujący instrukcję SWITCH..CASE najlepiej się przekonać na poniższym przykładzie.

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

using namespace std;

int main(int argc, char *argv[])
{
  int i;
  cout<<"Wprowadz liczbe\n";
  cin>>i;
  switch (i)
  {
   case  1:
        cout<<"jeden\n"; break;
   case  2:
       {
          cout<<"dwa\n";
          cout<<”Dwa jest parzyste\n”;
```

```
        }   break;
    case  3:
         cout<<"trzy\n";  break;
    default:
         cout<<"inna liczba\n";
   }
     system("PAUSE");
     return EXIT_SUCCESS;
}
```

**Program 21 - działanie instrukcji SWITCH..CASE**

Wczytujemy liczbę i jeśli jest ona z zakresu od 1 do 3, wypisujemy jej postać słowną. Jeśli ma ona inną wartość używamy polecenia występującego po `default`. Warto tu się przyjrzeć przypadkowi liczby 2. Kiedy chcemy użyć sekwencji poleceń, musimy je ująć w klamry. `Break` umieszczamy po klamrze zamykającej.

Instrukcje SWITCH..CASE można też zagnieżdżać jedna w drugiej (jak instrukcje IF).

## *Ćwiczenia*

1. Napisz program, który wczytuje liczbę i jeśli jest z przedziału:
   a) 1 do 6 to podaje jaka jest to ocena.
   b) od 1 do 7 to podaje jaki jest to dzień tygodnia.
   c) od 1 do 12 to podaje jaki jest to miesiąc.
2. Napisz program wczytujący 2 liczby i podający wynik jednego z czterech podstawowych działań matematycznych. Do wyboru działania użyj instrukcji SWITCH...CASE.
3. Napisz program wyznaczający pola figur geometrycznych: kwadratu, koła, trójkąta. Do wyboru figury użyj instrukcji SWITCH...CASE.
4. Przerób program wykorzystujący zagnieżdżoną instrukcję IF na SWITCH..CASE.
5. Napisz prosty test wyboru, wykorzystując instrukcję SWITCH..CASE.

# Pętle w C/C++

Pętla to jeden z najlepszym pomysłów w dziedzinie programowania. Pozwala nam na wykonanie całego zestawu instrukcji (takich samych lub podobnych). Operuje na takich samych operacjach, danych, strukturach, obiektach czy procedurach.
Korzyści widać wyraźnie przy porównaniu C/C++ z językami niestrukturalnymi (jak asembler). Wyobraźmy sobie, że mamy do wykonania np. 1000 razy operację przypisania wartości do jakiejś zmiennej. W asemblerze musielibyśmy taką instrukcję `Mov AX,BX` napisać 1000 razy[7]. Korzystając z pętli upraszczamy sprawę: `for (i=1; i<=1000;i++) AX=BX;`

W języku C/C++ mamy do dyspozycji dwie pętle: FOR i WHILE..DO.

## *Pętla FOR*

Pętla FOR jest tzw. pętlą licznikową. Oznacza to, że jej wykonywanie zależy od wartości licznika, czyli zmiennej sterującej działaniem pętli. FOR jest najpopularniejszą pętlą, używaną we wszystkich językach programowania (strukturalnych i obiektowych).

W języku C/C++ pętla FOR ma następującą postać:

```
for(inicjalizacja_licznika; warunek_pracy_pętli; zmiana_licznika)
```

`Inicjalizacja licznika` to wartość początkowa potrzebna do uruchomienia pętli FOR.

`Warunek pracy pętli` to wartość końcowa licznika, której przekroczenie kończy działanie pętli.

`Zmiana licznika` to instrukcje, które zmieniają wartość licznika. Najczęściej zwiększają go o 1, ale w C++ możemy w szeroki sposób tym sterować.

Po tym wstępie, zobaczmy przykład programu z pętlą.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int i;

    for (i=1; i<11; i++)
    {
        cout<<i<<endl;
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 22 - pętla FOR rosnąca**

[7] Mamy wprawdzie ulubione przez studentów polecenie **kopiuj – wklej**, ale to nie rozwiązuje problemu.

Program wyświetli nam liczby od 1 do 10.

Ten sam efekt da zmiana warunku na `for (i=1; i<=10; i++)`. Poniższe warunki są równoważne.

**Tabela 23 - równoważne warunki trwania pętli**

| i<11 | i<=10 | i!=11 |
|---|---|---|
| i>9 | i>=10 | i!=9 |

Zmieńmy teraz nasz program by wypisywał liczby w kolejności malejącej od 10 do 1.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int i;

    for (i=10; i>0; i--)
    {
        cout<<i<<endl;
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 23 - pętla FOR malejąca**

Jak widać, zmiana dotyczy tylko nagłówka pętli. Poprawiliśmy warunek początkowy (`i=10`), warunek pracy pętli (`i>0`) i operację zmiany licznika (`i--`). Zawartość pętli pozostała taka sama.

Oprócz dekrementacji i inkrementacji można użyć innych poleceń zmiany licznika. Poniżej zebrałem w tabelce kilka ciekawych, choć prostych operacji.

| | |
|---|---|
| `for (i=0; i<=100; i=i+2)` | Liczby parzyste z przedziału od 0 do 100 |
| `for (i=1; i<=100; i=i+2)` | Liczby nieparzyste z przedziału od 1 do 100 |
| `for (i=1; i<=100; i=i*2)` | Kolejne potęgi 2 z przedziału od 0 do 100 |

Słowo uwagi co do dwóch pierwszych pozycji. Jak widać, to te same pętle - różnią się tylko wartością początkową. Ponieważ zaczynają się od liczby parzystej (lub nieparzystej) i zwiększają się o 2, mamy tylko liczby jednego lub drugiego typu.

## Pętla nieskończona

W niektórych przypadkach chcemy, by pętla chodziła w kółko – była pętlą nieskończoną, przykładem takiej sytuacji jest program, który nadzoruje pracę jakiś urządzeń fabrycznych, czy zarządza siecią.

Najprostszym rozwiązaniem jest pętla z nagłówkiem bez zdefiniowanych parametrów. Ma ona postać

```
for(;;)
{
   jakieś_instrukcje_wykonywane_w_kółko;
}
```

Może też nie mieć tylko warunku końcowego. Efekt będzie ten sam.

```
for(i=1;;i++)
```

Czasem to samo osiąga się poprzez źle zaprojektowany nagłówek pętli.

```
for(i=0;<10;i--)
```

Ponieważ licznik się zmniejsza, a nie zwiększa to nigdy nie osiągnie warunku przerywającego działanie pętli.

Innym często spotykanym problemem jest nagłówek pętli FOR, która nigdy się nie uruchomi.

```
for(i=0;>10;i++)
```

Licznik zaczyna od zera i sprawdza, czy jest większy od 10. Ponieważ nie jest to możliwe, pętla nie zostanie uruchomiona.

## Wyjście z pętli nieskończonej

Jak więc sobie poradzić z taką pętlą[8]?

W języku C/C++ istnieje kilka sposobów rozwiązania tego typu problemów.

Najczęściej się używa instrukcji BREAK. Ma ona za zadanie natychmiastowe przerwanie działania pętli, bez sprawdzania jej warunku. Wykonywana jest następna instrukcja występująca za pętlą.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int i;

    for (i=0;i<10; i--)
    {
        cout<<i<<endl;
        if (i == -1000) break;
    }
    cout<<"nastepna instrukcja po break\n";
```

[8] Poza wyłączeniem komputera, oczywiście.

```
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 24 – Użycie instrukcji BREAK dla wyjścia z pętli FOR**

Drugim sposobem jest użycie instrukcji skoku GOTO. Stosuje się ją rzadko, ale czasem bywa przydatna. Wymaga zdefiniowana wcześniej etykiet w programie, między którymi może nawigować. Nadaje się świetnie do wychodzenia z wnętrza wielokrotnie zagnieżdżonych pętli.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int i,j,k;

    for (i=0;i<10; i--)
    {
       for (j=0;j<10; j--)
       {
          for (k=0;k<10; k--)
          {
             cout<<k<<endl;
             if (k == -1000) goto wyjscie;
          }
       }
    }
    wyjscie:
       cout<<"instrukcja po skoku goto\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 25 – Użycie instrukcji GOTO dla wyjścia z pętli FOR**

Jak widać zdefiniowaliśmy wcześniej etykietę `wyjscie`, która wskazuje odkąd ma być dalej wykonywany program[9].

W tym wypadku instrukcja BREAK nie byłaby dobra. Wyszlibyśmy tylko z pętli o najniższym poziomie. Ponieważ wszystkie pętle są nieskończone, musielibyśmy dodawać BREAK na każdym poziomie zagnieżdżenia lub rozbudować warunek wyjścia z pętli.

## Wielokrotne pętle FOR

Dowiedzieliśmy się już przy omówieniu GOTO, że istnieją pętle wielokrotne. Taka sekwencja zagnieżdżonych w sobie pętli ma postać:

```
for(i=1;i<=10;i++)
{
   for(j=1;j<=10;j++)
   {
```

[9] Ważne jest też odpowiednie umiejscowienie etykiety. Gdyby była ona wewnątrz tej samej pętli nic by nam to nie dało. Wykonywany wątek programu wróciłby w to samo miejsce.

```
        for (k=1;k<=10;k++)
        {
           for (l=1;l<=10;l++)
           {
                 jakieś_instrukcje_wykonywane_w_pętlach;
           }
        }
    }
}
```

Instrukcje wewnątrz pętli są wykonywane tyle razy ile wynosi iloczyn liczników pętli wyższego poziomu. W powyższym przykładzie są wykonywane 10*10*10*10 razy, czy 10 000 razy.

Przykład takiego programu poniżej. Wypisuje ona na ekranie zawartość tabliczki mnożenia z zakresu 1-100.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int i,j;

    for (i=1;i<=10; i++)
    {
       for (j=1;j<=10; j++)
       {
          printf("%4d",i*j);          //wyświetlana zmienna
       }
       cout<<endl;     //koniec linii
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 26 – przykład wielokrotnej pętli FOR**

Program zawiera dwie pętle FOR (jedna w drugiej). Wykonywanym poleceniem jest wypisywanie iloczynu na ekranie (tak działa tabliczka mnożenia). Jednak posłużyliśmy się inna instrukcją. To `printf`. Służy ona do wyświetlania tekstu na ekranie, a więc realizuje to samo co `cout`. Ma jednak większe możliwości konfigurowania sposobu i rodzaju wyświetlanych danych. W nawiasie podajemy najpierw opcje wyświetlania, a potem (po przecinku) wyświetlaną zmienną.

W powyższym przykładzie opcja `"%d"`, oznacza liczbę całkowitą. Czwórka stojąca przed nią `"%4d"`, informuje, że trzeba na wyświetlenie liczby zarezerwować cztery znaki[10]. Wyświetlaną zmienną jest iloczyn `i*j`.

[10] Wynika to stąd, że największa liczba 100 ma trzy cyfry, a jeszcze trzeba uwzględnić odstęp pomiędzy liczbami, by uzyskać czytelny efekt.

W pętli głównej jest dopisana instrukcja zakończenia linii `cout<<endl`. Realizowana ona jest po zakończeniu pojedynczego wiersza tabliczki mnożenia. Dlatego jest ona wykonywana po zakończeniu pętli wewnętrznej.

## *Pętla WHILE*

W języku C/C++ istnieje druga pętla – WHILE. Jest to pętla warunkowa, która jest wykonywana, aż do odpowiedniej zmiany warunków. Stąd pochodzi jej nazwa – *While*, czyli *dopóki*. Dopóki warunek jest spełniony, instrukcje są wykonywane.

Argumentami pętli WHILE nie muszą być tylko liczby. Może to być dowolny warunek. Przeważnie sprawdzany jest na początku pętli.

W języku C/C++ pętla WHILE ma następującą postać:

```
while(warunek_pracy_pętli)
{
   wykonywane_instrukcje
}
```

`Warunek pracy pętli` to wartość wyrażenia, którego spełnienie kończy działanie pętli. Jeśli jest prawdziwy, wykonywane są instrukcje. Jeśli nie, pętla może nie zostać wykonana ani razu.

Budowa pętli jest więc prosta. Zobaczmy więc teraz przykład programu.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int i;
    i=1;              //inicjalizacja zmiennej, która jest licznikiem pętli
    while(i<11)       //sprawdzenie warunku pętli
    {
        cout<<i<<endl;   //wykonywana operacja
        i++;             //zmiana licznika pętli
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 27 - pętla WHILE rosnąca**

Program również wyświetli nam liczby od 1 do 10. Ale pętla WHILE wymaga od nas pamiętania o pewnych rzeczach. Musimy przed pętlą zainicjować zmienną, która ma być warunkiem pętli `i=1`. Przy pętli WHILE musimy pamiętać o liczniku pętli.

Również nie można zapomnieć o inkrementacji licznika. Musimy zadbać o to sami. Dlatego trzeba odpowiednią instrukcję `i++` umieścić wewnątrz pętli.

Zmieńmy teraz nasz program by wypisywał liczby w kolejności malejącej od 10 do 1.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int i;
    i=10;                    //nowa wstępna wartość licznika pętli
    while(i>0)          //Zmieniony warunek pętli
    {
       cout<<i<<endl;
       i--;                  //dekrementacja licznika pętli
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 28 - pętla WHILE malejąca**

Jak widać, poprawiliśmy warunek początkowy (`i=10`) i warunek pracy pętli (`i>0`). Wykonywane w pętli instrukcje pozostały bez zmian, ale korekta dotyczyła licznika (`i--`).

## Pętla warunkowa

Oprócz typowych operacji związanych z liczbami, możemy też użyć go do sprawdzania czy jakiś warunek nieliczbowy jest prawdziwy.

Poniżej podano przykład wczytywania znaków z klawiatury aż do chwili gdy wczytamy literę ‘n’.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    char x;
    while (x!='n')                 //warunek pętli
    {
       cout<<"podaj znak\n";
       cin>>x;                 //wczytanie znaku
    }
                                   //instrukcja wykonywana po zakończeniu pętli
    cout<<"Wybranie n konczy petle\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 29 - pętla warunkowa do rozpoznawania znaków z klawiatury**

W pętli wczytujemy pojedyncze znaki z klawiatury i sprawdzamy, czy są równe ‘n’. Jeśli tak, to kończymy działanie pętli.

Taka konstrukcja jest przydatna przy tworzeniu interfejsu programu. Często chcemy, by program chodził dopóki nie naciśniemy odpowiedniego klawisza (lub ich kombinacji). Wtedy korzystne jest użycie powyższej konstrukcji wykorzystujące WHILE.

## Pętla nieskończona

Podobna konstrukcją jest pętla nieskończona. Ta jednak chodzi cały czas. W odróżnieniu od FOR, nie ma specjalnej formy nieskończonej pętli WHILE. Należy tutaj ustawić odpowiedni warunek.

```
i=1;
while(i<10)
{
 i--;
}
```

Ponieważ licznik się zmniejsza nigdy nie osiągnie warunku przerywającego pętlę.
Analogicznie można stworzyć nagłówek pętli, która nigdy się nie uruchomi.

```
i=1;
while(i>10)
{
 i++;
}
```

Licznik zaczyna od zera i sprawdza, czy jest większy od 10. Ponieważ nie jest to możliwe, pętla nie zostanie uruchomiona.

## Wyjście z pętli nieskończonej

Tu też przydatna jest instrukcja BREAK.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int i;
    i=1;
    while (i<=10)
    {
       cout<<i<<endl;
       i--;
       if  (i<-1000) break;
    }
    cout<<"nastepna instrukcja po break\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 30 – Użycie instrukcji BREAK dla wyjścia z pętli FOR**

Drugim sposobem jest użycie instrukcji skoku GOTO.

```
#include <cstdlib>
#include <iostream>

using namespace std;
```

```
int main(int argc, char *argv[])
{
    int i,j,k;
    i=1;
    while (i<=10)
    {
       j=1;
       while (j<=10)
       {
           k=1;
           while (k<=10)
          {
              cout<<k<<endl;
              k--;
              if  (k<-1000) goto wyjscie;
          }
          j--;
       }
       i--;
    }
    wyjscie:
       cout<<"instrukcja po skoku goto\n";

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 31 – Użycie instrukcji GOTO dla wyjścia z pętli FOR**

## Wielokrotne pętle WHILE

Pętla WHILE również może być użyta jedna w drugiej. Taka sekwencja zagnieżdżonych w sobie pętli ma postać:

```
i=1;
while(i<=10)
{
   j=1;
   while(j<=10)
   {
       k=1;
       while (k<=10)
       {
          l=1;
          while (l<=10)
          {
                jakieś_instrukcje_wykonywane_w_pętlach;
                l++;
          }
          k++;
       }
       j++;
   }
   i++;
```

```
}
```

Widać jednak, że użycie ich bardziej komplikuje wygląd programu. Należy przed każdą pętlą zainicjować licznik dla niej. Nie można tego zrobić przed tą najbardziej zewnętrzną, gdyż każda z nich musi być wykonywana tyle razy ile występuje pętli nadrzędnych. Należy też uwzględnić konieczność modyfikacji licznika danej pętli (w powyższym przykładzie umieściłem stosowne instrukcje po wykonaniu wewnętrznej pętli).

Przykład takiego programu poniżej. Wypisuje ona na ekranie zawartość tabliczki mnożenia z zakresu 1-100.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int i,j;
    i=1;
    while (i<=10)
    {
       j=1;
       while (j<=10)
       {
          printf("%4d",i*j);      //wyświetlana zmienna
          j++;
       }
       i++;
       cout<<endl;   //koniec linii
    }

    system("PAUSE");
    return EXIT_SUCCESS;
};
}
```

**Program 32 – przykład wielokrotnej pętli WHILE**

## *Pętla DO..WHILE*

W języku C/C++ istnieje jeszcze odmiana pętli WHILE. To pętla DO..WHILE.

Jest to pętla podobna w działaniu do pętli WHILE tzn. jeśli warunek jest spełniony, instrukcje w pętli są wykonywane. Jednak w przeciwieństwie do niej, operacje pomiędzy słowami kluczowymi DO i WHILE są wykonywane co najmniej jeden raz[11]. Wynika to z tego, że sprawdzanie warunku odbywa się na końcu pętli (a nie na jej początku jak dla FOR czy WHILE).

W języku C/C++ pętla DO..WHILE ma następującą postać:

[11] Jest więc podobna w działaniu do pętli REPEAT z Pascala.

```
do
{
   wykonywane_instrukcje

}
while(warunek_pracy_pętli)
```

`Warunek pracy pętli` to wartość wyrażenia, którego spełnienie kończy działanie pętli. Jeśli jest prawdziwy, wykonywane są instrukcje. Jeśli nie, pętla po pierwszym przebiegu kończy pracę.

Pętla DO..WHILE jest rzadko używana. Zawsze istnieje możliwość takiego ustawienia warunków pętli WHILE (lub FOR) by wykonywała się co najmniej jeden raz.

Budowa pętli też jest prosta. Zobaczmy teraz przykład programu.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int i;
    i=1;                //inicjalizacja zmiennej, która jest licznikiem pętli
    do              //rozpoczęcie petli
    {
        cout<<i<<endl;        //wykonywana operacja
        i++;                //zmiana licznika pętli
    }
    while(i<11);              //sprawdzenie warunku pętli

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 33 - pętla DO..WHILE rosnąca**

Program wyświetli nam liczby od 1 do 10. Analogicznie do WHILE, pętla DO..WHILE również wymaga pamiętania o pewnych rzeczach. Musimy przed pętlą zainicjować warunek pętli `i=1` i nie można zapomnieć o inkrementacji licznika wewnątrz pętli.

Zmieńmy teraz nasz program by wypisywał liczby w kolejności malejącej od 10 do 1.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int i;
    i=10;                //inicjalizacja zmiennej, która jest licznikiem pętli
    do              //rozpoczęcie petli
```

```
    {
        cout<<i<<endl;       //wykonywana operacja
        i--;                 //zmiana licznika pętli
    }
    while(i>0);              //sprawdzenie warunku pętli

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 34 - pętla DO..WHILE malejąca**

Jak widać, poprawiliśmy warunek początkowy (`i=10`) i warunek pracy pętli (`i>0`). Wykonywane w pętli instrukcje pozostały bez zmian, ale korekta dotyczyła licznika (`i--`).

## Pętla warunkowa

Również pętle DO..WHILE możemy użyć do sprawdzania nieliczbowego warunku. Jeszcze raz zobaczmy przykład wczytywania znaków z klawiatury aż do wczytania litery ‘n’.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    char x;
   do                        //inicjalizacja pętli
   {
      cout<<"podaj znak\n";
      cin>>x;                //wczytanie znaku
    }   while (x!='n')                   //warunek pętli

                                   //instrukcja wykonywana po zakończeniu pętli
    cout<<"Wybranie n konczy petle\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 35 - pętla warunkowa DO..WHILE do rozpoznawania znaków z klawiatury**

Pętla działa analogicznie do WHILE.

Podobieństw w budowie i działaniu do WHILE jest więcej. Nie będę ich tu opisywał. Przeróbkę odpowiednich programów zostawiam czytelnikowi jako ćwiczenie.

**Tabela 24 - Porównanie pętli FOR, WHILE, DO..WHILE w C/C++**

<table>
<tr><td><pre>for (i=1; i<11; i++)
{
 cout<<i<<endl;
}</pre></td><td><pre>i=1;
while(i<11)
{
 cout<<i<<endl;
 i++;
}</pre></td><td><pre>i=1;
do
{
 cout<<i<<endl;
 i++;
}
while(i<11);</pre></td></tr>
<tr><td colspan="3">Pętla wypisująca od 1 do 10</td></tr>
<tr><td><pre>for (i=10; i>0; i--)</pre></td><td><pre>i=10;</pre></td><td><pre>i=10;</pre></td></tr>
</table>

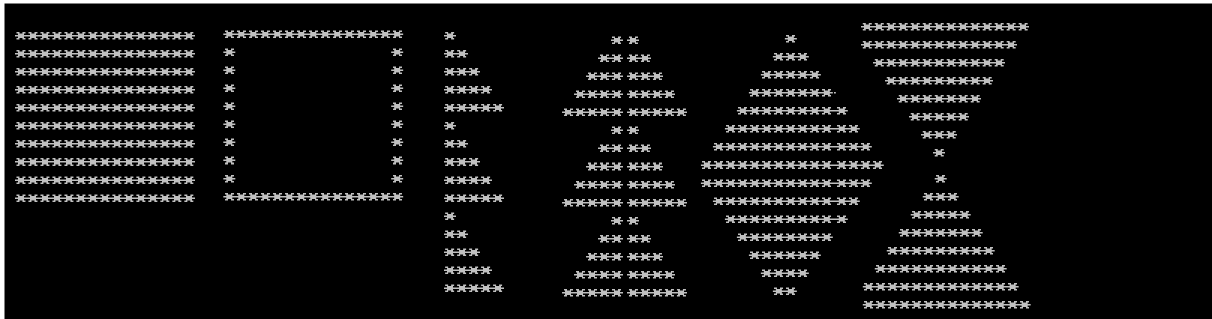
| | | |
|---|---|---|
| {<br> cout<<i<<endl;<br>} | while(i>0)<br>{<br> cout<<i<<endl;<br> i--;<br>} | do<br>{<br> cout<<i<<endl;<br> i--;<br>}<br>while(i>0); |
| Pętla wypisująca od 10 do 1 | | |
| for (i=1;i<=10; i++)<br>{<br> for(j=1;j<=10; j++)<br> {<br>  printf("%4d",i*j);<br> }<br> cout<<endl;<br>} | i=1;<br>while (i<=10)<br>{<br> j=1;<br> while (j<=10)<br> {<br>  printf("%4d",i*j);<br>  j++;<br> }<br> i++;<br>       cout<<endl;<br>} | i=1;<br>do<br>{<br> j=1;<br> do<br> {<br>  printf("%4d",i*j);<br>  j++;<br> }<br> while (j<=10);<br> i++;<br> cout<<endl;<br>}<br>while (i<=10); |
| Pętla wielokrotna | | |
| for(i=0;i<10;i--)<br>{<br><br>} | i=1;<br>while(i<10)<br>{<br> i--;<br>} | i=1;<br>do<br>{<br> i--;<br>}<br>while(i<10) |
| Pętla nieskończona | | |

Porównując powyższe przykłady widzimy, że najbardziej czytelna jest pętla FOR. Nic więc dziwnego, że stosuje się ją najczęściej. Zgromadzenie wszystkich poleceń sterujących w nagłówku pętli jest dobrym rozwiązaniem. Przydaje się to zwłaszcza przy kilku pętlach zagnieżdżonych w sobie.

## *Ćwiczenia*

1. Napisz program wyświetlający tabliczkę mnożenia w zakresie od 1 do 10
2. Napisz program, który wczytuje liczbę **n**, a następnie sumuje **n** liczb, wczytywanych
3. z klawiatury.
4. Napisz program obliczający wartość średniej arytmetycznej z podanych n liczb.
5. Napisz program, który wczytuje liczbę n, a następnie wylicza sumę liczb od 1 do n.
6. Napisz program, który wczytuje liczbę **n**, a następnie wylicza silnię **n!**
7. Napisz program wyświetlający kwadraty i sześciany liczb od 1 do 100.
8. Napisz program rysujący pełny kwadrat złożony z gwiazdek. Ilość gwiazdek (czyli długość boku) podajemy z klawiatury.
9. Napisz program rysujący pusty kwadrat otoczony ramką z gwiazdek. Ilość gwiazdek (czyli długość boku) podajemy z klawiatury.
10. Napisz program rysujący choinkę jednostronną. Wczytujemy ilość trójkątów i ilość wierszy w każdym z nich.
11. Napisz program rysujący choinkę dwustronną. Wczytujemy ilość trójkątów i ilość wierszy w każdym z nich.

12. Napisz program rysujący romb złożony z gwiazdek. Ilość gwiazdek (czyli długość boku) podajemy z klawiatury.
13. Napisz program rysujący klepsydrę złożoną z gwiazdek. Ilość gwiazdek (czyli długość boku pojedynczej części) podajemy z klawiatury.



14. Napisz program rysujący równoległobok złożony z gwiazdek. Ilość gwiazdek (czyli długość boków) podajemy z klawiatury.
15. Napisz program wyświetlający szachownicę.

# Tablice w C/C++

Zmienne w języku C/C++ to przydatna rzecz. Jednak problem z ich używaniem pojawia się wtedy gdy mamy większy zbiór danych do analizy. Wyobraźmy sobie, że musimy przebadać 1000 zmiennych. Jeśli chcielibyśmy je przeanalizować oddzielnie, musielibyśmy ręcznie utworzyć deklaracje ich wszystkich, pamiętać o ich aktualizacji, działaniach na nich i wyświetlaniu. Oczywiście jest to możliwe, ale bardzo żmudne w realizacji i czasochłonne.

Żeby czas spożytkować na lepsze i ciekawsze zadania wymyślono tablice. Ich ogólna zasada polega na tym, że zmienne są zebrane razem i traktowane w jednakowy sposób. Normalnie byłyby rozproszone, a tak tworzą jedną całość.

| Tablica to spójny zbiór danych jednakowego typu |
|---|

Tablice są używane bardzo chętnie przez programistów.

W rozdziale Zmienne w C dowiedziałeś się, jak przechowywać pojedyncze liczby oraz znaki. Czasami zdarza się jednak, że potrzebujemy przechować kilka, kilkanaście albo i więcej zmiennych jednego typu. Nie tworzymy wtedy np. dwudziestu osobnych zmiennych. W takich przypadkach z pomocą przychodzi nam **tablica**.

Tablica to ciąg zmiennych jednego typu. Ciąg taki posiada jedną nazwę a do jego poszczególnych elementów odnosimy się przez numer (indeks).

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

tablica 10-elementowa

## *Spis treści*


- 1 Wstęp
  - 1.1 Sposoby deklaracji tablic
- 2 Odczyt/zapis wartości do tablicy
- 3 Tablice znaków
- 4 Tablice wielowymiarowe
  - 4.1 Kolejność głównych wierszy
- 5 Ograniczenia tablic
- 6 Zobacz również
- 7 Przypisy


## *Wstęp*

### Sposoby deklaracji tablic

Tablicę deklaruje się w następujący sposób:

```
typ nazwa_tablicy[rozmiar];
```

gdzie rozmiar oznacza ile zmiennych danego typu możemy zmieścić w tablicy. Zatem aby np. zadeklarować tablicę, mieszczącą 20 liczb całkowitych możemy napisać tak:

```
int tablica[20];
```

Podobnie jak przy deklaracji zmiennych, także tablicy możemy nadać wartości początkowe przy jej deklaracji. Odbywa się to przez umieszczenie wartości kolejnych elementów oddzielonych przecinkami wewnątrz nawiasów klamrowych:

```
int tablica[3] = {0,1,2};
```

Niekoniecznie trzeba podawać rozmiar tablicy, np.:

```
int tablica[] = {1, 2, 3, 4, 5};
```

W takim przypadku kompilator sam ustali rozmiar tablicy (w tym przypadku - 5 elementów).

Rozpatrzmy następujący kod:

```
#include <stdio.h>
#define ROZMIAR 3
int main()
{
  int tab[ROZMIAR] = {3,6,8};
  int i;
  puts ("Druk tablicy tab:");

  for (i=0; i<ROZMIAR; ++i) {
    printf ("Element numer %d = %d\n", i, tab[i]);
  }
  return 0;
}
```

Wynik:

```
Druk tablicy tab:
Element numer 0 = 3
Element numer 1 = 6
Element numer 2 = 8
```

Jak widać, wszystko się zgadza.

W powyżej zamieszczonym przykładzie użyliśmy stałej do podania rozmiaru tablicy. Jest to o tyle pożądany zwyczaj, że w razie potrzeby zmiany rozmiaru tablicy, zmieniana jest tylko wartość w jednej linijce kodu przy `#define`, w innym przypadku musielibyśmy szukać wszystkich wystąpień rozmiaru rozsianych po kodzie całego programu.

## *Odczyt/zapis wartości do tablicy*

Tablicami posługujemy się tak samo jak zwykłymi zmiennymi. Różnica polega jedynie na podawaniu **indeksu** tablicy. Określa on, z którego elementu (wartości) chcemy skorzystać

spośród wszystkich umieszczonych w tablicy. Numeracja indeksów rozpoczyna się od zera, co oznacza, że pierwszy element tablicy ma indeks równy 0, drugi 1, trzeci 2, itd.

**Uwaga!**

Osoby, które wcześniej programowały w językach, takich jak Pascal, Basic czy Fortran, muszą przyzwyczaić się do tego, że w języku C indeks numeruje się od 0.

Spróbujmy przedstawić to na działającym przykładzie. Przeanalizuj następujący kod:

```
int tablica[5] = {0};
int i = 0;
tablica[2] = 3;
tablica[3] = 7;
for (i=0;i!=5;++i) {
  printf ("tablica[%d]=%d\n", i, tablica[i]);
}
```

Jak widać, na początku deklarujemy 5-elementową tablicę, którą od razu zerujemy. Następnie pod trzeci i czwarty element (liczone począwszy od 0) podstawiamy liczby 3 i 7. Pętla ma za zadanie wyprowadzić wynik naszych działań.

## *Tablice znaków*

Tablice znaków, tj. typu char oraz unsigned char, posiadają dwie ogólnie przyjęte nazwy, zależnie od ich przeznaczenia:

- bufory - gdy wykorzystujemy je do przechowywania ogólnie pojętych danych, gdy traktujemy je jako po prostu "ciągi bajtów" (typ char ma rozmiar 1 bajta, więc jest elastyczny do przechowywania np. danych wczytanych z pliku przed ich przetworzeniem).
- napisy - gdy zawarte w nich dane traktujemy jako ciągi liter; jest im poświęcony osobny rozdział Napisy.

## *Tablice wielowymiarowe*

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | | | | | |
| 1 | | | | | |
| 2 | | | | | |
| 3 | | | | | |
| 4 | | | | | |

tablica dwuwymiarowa (5x5)

Rozważmy teraz konieczność przechowania w pamięci komputera całej macierzy o wymiarach 10 x 10. Można by tego dokonać tworząc 10 osobnych tablic jednowymiarowych, reprezentujących poszczególne wiersze macierzy. Jednak język C dostarcza nam dużo wygodniejszej metody, która w dodatku jest bardzo łatwa w użyciu. Są to **tablice**

**wielowymiarowe**, lub inaczej "tablice tablic". Tablice wielowymiarowe definiujemy podając przy zmiennej kilka wymiarów, np.:

```
float macierz[10][10];
```

Tak samo wygląda dostęp do poszczególnych elementów tablicy:

```
macierz[2][3] = 1.2;
```

Jak widać ten sposób jest dużo wygodniejszy (i zapewne dużo bardziej "naturalny") niż deklarowanie 10 osobnych tablic jednowymiarowych. Aby zainicjować tablicę wielowymiarową należy zastosować zagłębianie klamr, np.:

```
float macierz[3][4] = {
  { 1.6, 4.5, 2.4, 5.6 },  /* pierwszy wiersz */
  { 5.7, 4.3, 3.6, 4.3 },  /* drugi wiersz */
  { 8.8, 7.5, 4.3, 8.6 }   /* trzeci wiersz */
};
```

Dodatkowo, pierwszego wymiaru nie musimy określać (podobnie jak dla tablic jednowymiarowych) i wówczas kompilator sam ustali odpowiednią wielkość, np.:

```
float macierz[][4] = {
  { 1.6, 4.5, 2.4, 5.6 },  /* pierwszy wiersz */
  { 5.7, 4.3, 3.6, 4.3 },  /* drugi wiersz */
  { 8.8, 7.5, 4.3, 8.6 },  /* trzeci wiersz */
  { 6.3, 2.7, 5.7, 2.7 }  /* czwarty wiersz */
};
```

Innym, bardziej elastycznym sposobem deklarowania tablic wielowymiarowych, jest użycie wskaźników. Opisane to zostało w następnym rozdziale.

## Kolejność głównych wierszy

Kolejność głównych wierszy ( ang. Row Major Order = ROM [1])

W C tablica wielowymiarowa A[n][m] :

- jest przechowywana wierszami[2] :
- numeracja indeksów rozpoczyna się od zera

```
A[0][0], A[0][1], ..., A[0][m-1], A[1][0], A[1][1],..., A[n-1][m-1]
```

Przykładowy program :

```
/*
 http://stackoverflow.com/questions/2151084/map-a-2d-array-onto-a-1d-array-c/2151113

*/
  #include <stdio.h>
```

```
int main(int argc, char **argv) {
int i, j, k;
int arr[5][3];
int *arr2 = (int*)arr;

    for (k=0; k<15; k++) {
       arr2[k] = k;
       printf("arr[%d] = %2d\n", k, arr2[k]);
    }

    for (i=0; i<5; i++) {
      for (j=0; j< 3; j++) {
         printf("arr2[%d][%d] = %2d\n", i, j ,arr[i][j]);
      }
    }
 }
```

## *Ograniczenia tablic*

Pomimo swej wygody **tablice statyczne** mają ograniczony, z góry zdefiniowany rozmiar, którego nie można zmienić w trakcie działania programu. Dlatego też w niektórych zastosowaniach tablice statyczne zostały wyparte **tablicami dynamicznymi**, których rozmiar może być określony w trakcie działania programu. Zagadnienie to zostało opisane w następnym rozdziale.

**Uwaga!**

Przy używaniu tablic trzeba być szczególnie ostrożnym przy konstruowaniu pętli, ponieważ ani kompilator, ani skompilowany program nie będą w stanie wychwycić przekroczenia przez indeks rozmiaru tablicy [3]. Efektem będzie odczyt lub zapis pamięci, znajdującej się poza tablicą.

Programiści C++ mogą użyć klasy vector, która może być wygodnym zamiennikiem tablic.

Wystarczy pomylić się o jedno miejsce (tzw. błąd off by one) by spowodować, że działanie programu zostanie nagle przerwane przez system operacyjny:

```
int foo[100];
int i;

for (i=0; i<=100; i+=1) /* powinno być
```

Wskaźniki

```
#include <stdio.h>

 int main (void)
 {
   int liczba = 80;
   printf("Wartość zmiennej liczba: %d\n", liczba );
   printf("Adres zmiennej liczba: %p\n", &liczba );
   return 0;
 }
```

```
int *wskaznik1;    // zmienna wskaźnikowa na obiekt typu liczba całkowita
char *wskaznik2;    // zmienna wskaźnikowa na obiekt typu znak
float *wskaznik3;// zmienna wskaźnikowa na obiekt typu liczba
zmiennoprzecinkowa
```

Niektórzy programiści mogą nieco błędnie interpretować wskaźnik do typu jako nowy typ i uważać, że jeśli napiszą:

```
 int * a,b,c;
```

to otrzymają trzy wskaźniki do liczby całkowitej. W rzeczywistości uzyskamy jednak tylko jeden wskaźnik a, oraz dwie liczby całkowite b i c (tak jakbyśmy napisali int *a; int b, int c). W tym przypadku trzy wskaźniki otrzymamy pisząc:

```
 int *a,*b,*c;
```

Aby uniknąć pomyłek, lepiej jest pisać gwiazdkę tuż przy zmiennej, albo jeszcze lepiej - nie mieszać deklaracji wskaźników i zmiennych:

```
 int *a;
 int b,c;
```

```
#include <stdio.h>

 int main (void)
 {
   int liczba = 80;
   int *wskaznik = &liczba;   // wskaznik przechowuje adres, ktory
pobieramy od zmiennej liczba

   printf("Wartosc zmiennej: %d, jej adres: %p.\n", liczba, &liczba);
   printf("Adres przechowywany we wskazniku: %p, wskazywana wartosc:
%d.\n",
          wskaznik, *wskaznik);
```

```
   *wskaznik = 42;   // zapisanie liczby 42 do obiektu, na który wskazuje
wskaznik
   printf("Wartosc zmiennej: %d, wartosc wskazywana przez wskaznik: %d\n",
          liczba, *wskaznik);

   liczba = 0x42;
   printf("Wartosc zmiennej: %d, wartosc wskazywana przez wskaznik: %d\n",
          liczba, *wskaznik);

   return 0;
 }
```

## *Stałe wskaźniki*

Podobnie jak możemy deklarować zwykłe stałe, tak samo możemy mieć stałe wskaźniki - jednak są ich dwa rodzaje. Wskaźniki na stałą wartość:

```
const int *a;
int const * a;  /* równoważnie */
```

oraz stałe wskaźniki:

```
int * const b;
```

Słówko `const` przed typem działa jak w przypadku zwykłych stałych, tzn. nie możemy zmienić wartości wskazywanej przy pomocy wskaźnika.

W drugim przypadku słowo `const` jest tuż za gwiazdką oznaczającą typ wskaźnikowy, co skutkuje stworzeniem stałego wskaźnika, czyli takiego którego nie można przestawić na inny adres.

Obie opcje można połączyć, deklarując stały wskaźnik, którym nie można zmienić wartości wskazywanej zmiennej, i również można zrobić to na dwa sposoby:

```
const int * const c;
int const * const c;  /* równoważnie  */

int i=0;
const int *a=&i;
int * const b=&i;
int const * const c=&i;
*a = 1;  /* kompilator zaprotestuje */
*b = 2;  /* ok */
*c = 3;   /* kompilator zaprotestuje */
a = b;   /* ok */
b = a;   /* kompilator zaprotestuje */
c = a;   /* kompilator zaprotestuje */
```

Wskaźniki na stałą wartość są przydatne między innymi w sytuacji gdy mamy duży obiekt (na przykład strukturę z kilkoma polami). Jeśli przypiszemy taką zmienną do innej zmiennej, kopiowanie może potrwać dużo czasu, a oprócz tego zostanie zajęte dużo pamięci.

Przekazanie takiej struktury do funkcji albo zwrócenie jej jako wartość funkcji wiąże się z takim samym narzutem. W takim wypadku dobrze jest użyć wskaźnika na stałą wartość.

```
void funkcja(const duza_struktura *ds)
{
   /* czytamy z ds i wykonujemy obliczenia */
}
....
funkcja(&dane); /* mamy pewność, że zmienna dane nie zostanie zmieniona */
```

# Łańcuchy tekstowe

W języku C i C++ nie ma specjalnego typu do oznaczania łańcucha tekstowego. Zastosowano tu inne rozwiązanie. W pamięci mamy ciąg znaków ASCII w kolejnych komórkach pamięci, zakończonych znakiem pustym (*null character*).

| Łańcuch tekstowy to tablica znaków, zakończona znakiem NULL `'\0'` |
|---|

Zapis łańcucha od pojedynczego znaku wyróżnia się sposobem zapisu. Znak jest ujęty w apostrofy, a łańcuch tekstowy w cudzysłowy.

| | **Ilość znaków** | **Ilość bajtów** | **Zawartość** |
|---|---|---|---|
| `'a'` | Jeden znak | Jeden bajt | 97 (kod ASCII *litery a*) |
| `"a"` | Dwa znaki | Dwa bajty | 97 i 0 (kody ASCII *litery a* i *znaku pustego*) |

Jak wypisać łańcuch tekstowy na ekranie?

Używając instrukcji `cout` jest to bardzo proste. Tekst wypisujemy w cudzysłowach, a zmienna tekstową bez żadnych dodatkowych parametrów.

```
cout<<"Spory tekst do wypisania na ekranie komputera";
cout<<s1;
```

Używając funkcji `printf` musimy zaznaczyć, że chodzi o łańcuch tekstowy. Należy więc zastosować parametr **%s**.

```
Printf ("%s", "Spory tekst do wypisania");
Printf ("%s", s1);
```

Czas teraz na przyjrzenie się, jak wygląda gotowy program operujący na łańcuchach tekstowych.

```
#include <cstdlib>
#include <iostream>
#include <stdio.h>

using namespace std;
int main(int argc, char *argv[])
{
    char s1[20];

    cin>>s1;
    cout<<s1<<"\n";

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

**Program 36 - program operujący na łańcuchu tekstowym**

## *Operacje na łańcuchach tekstowych*

Język C/C++ posiada funkcje umożliwiające różne operacje na łańcuchach tekstowych. Zestawienie najpopularniejszych znajduje się w tabelce poniżej.

**Tabela 25 - Najpopularniejsze operacje na łańcuchach tekstowych**

| **Kopiowanie Łańcuchów** | |
|---|---|
| `strcpy (str, „Witam”);` | do zmiennej **str** kopiujemy tekst **Witamy** |
| `strcpy (str, s1);` | do zmiennej **str** kopiujemy zawartość zmiennej **s1** |
| **Sklejanie Łańcuchów** | |
| `strcat (str, „Witam”);` | sklejenie zmiennej **str** i tekstu **Witamy** |
| `strcat (str, s1);` | sklejenie zmiennej **str** i zmiennej **s1** |
| **Porównanie Łańcuchów** | |
| `x =  strcmp (str, „Witam”);` | porównanie zmiennej **str** i tekstu **Witamy** |
| `x =  strcmp (str, s1);` | porównanie zmiennej **str** i zmiennej **s1** |
| `x>0`<br>`x==0`<br>`x<0`<br>`if x==0  cout <<”rowne”`<br>`else cout<<”Nierowne”;` | str mniejsze od s1<br>str równe s1<br>str większe od s1 |
| **Odwrócenie  Łańcuchów** | |
| `strrev (str);` | odwrócenie zmiennej **str** |
| **Długość Łańcucha** | |
| `int x;`<br>`x=strlen (str);`<br>`cout<<x;` | długość zmiennej **str** |
| **Pozycja znaku Łańcucha** | |
| `char *p;`<br>`p=strchr (str,c);`<br>`cout<<p;` | zwraca miejsce znaku w zmiennej **str** |

## *Ćwiczenia*

1. Napisz program wypisujący tylko parzyste
2. Napisz program sprawdzający czy dany łańcuch tekstowy jest palindromem.
3. Napisz program, który szyfruje i deszyfruje dany tekst, używając szyfru cezara.
4.

Napisz program, który wczytuje imię (lub nazwisko) i podaje z ilu liter się składa.

Napisz program, który wczytuje dwa ciągi tekstowe i łączy je w jeden ciąg. Dodaj możliwość wyboru, który ma być pierwszy, a który drugi.

Napisz program, który dzieli dany ciąg tekstowy na dwa różne w zależności od wyboru miejsca podziału.

Napisz program wyszukujący w danym ciągu tekstowym zadaną frazę.

Napisz program wykasowujący w danym ciągu tekstowym zadaną frazę.

Napisz program zamieniający w danym ciągu tekstowym zadaną frazę na inną.

Napisz program rozdzielający litery z danego tekstu: parzyste do jednego, nieparzyste do drugiego.

Napisz program sklejający litery do danego tekstu: parzyste z jednego, nieparzyste z drugiego.

Napisz program, który wczytuje dany tekst i przepisuje go od tyłu.

Napisz program porównujący dwa teksty i informujący, czy są identyczne.

W danym łańcuchu zamienić wszystkie spacje na znaki podkreślenia.

Obliczyć ilość wystąpień danego znaku w łańcuchu.

Usuń wszystkie spacje z danego łańcucha.

Usuń, co drugi znak w danym łańcuchu.

Wstaw spację, w co drugi znak danego łańcucha.

Napisz program, który wczytuje dwie liczby jako zmienne tekstowe i przekształca je na liczbę. Następnie oblicz ich sumę/iloczyn.

Napisz program wczytujący łańcuch tekstowy. Sprawdza czy jest palindromem. Jeśli jest wypisuje komunikat że to palindrom. Jeśli nie skleja wczytany łańcuch i jego odwrotność w jeden nowy. Następnie wyświetla jego wszystkie nieparzyste znaki.

Oceniana jest dokładność wykonania zadania, czytelność programu i jednoznaczność komunikatów programu.

Napisz program wczytujący 4 łańcuchy tekstowe. Następnie skleja je w jeden wielki. Program umożliwia wybór kolejności sklejenia łańcuchów. Po sklejeniu ich wyświetla informację jak długi jest nowo powstały łańcuch.

Oceniana jest dokładność wykonania zadania, czytelność programu i jednoznaczność komunikatów programu.

Napisz program wczytujący 2 łańcuchy tekstowe. Następnie wyświetla kolejno na ekranie parzyste litery jednego łańcucha i nieparzyste z drugiego. Po zakończeniu wyświetlania podaje ile znaków wyświetlił.

Oceniana jest dokładność wykonania zadania, czytelność programu i jednoznaczność komunikatów programu.

Napisz program wczytujący łańcuch tekstowy. Następnie wyświetla tylko litery **a**, podając jednocześnie jaki jest numer danego znaku w łańcuchu tekstowym. Na końcu podaje informację ile było liter **a** w tekście.

Oceniana jest dokładność wykonania zadania, czytelność programu i jednoznaczność komunikatów programu.

# Funkcje matematyczne

# Operacje bitowe

Operatory bitowe operują na postaci bitowej liczb całkowitych. Są one szybsze od innych operacji.

Tabela 26 - operatory bitowe

<table>
<tr><td>& &=</td><td>Iloczyn bitowy (AND)</td><td>a = b & c</td></tr>
<tr><td>| |=</td><td>Suma bitowa (OR)</td><td>a = b | c</td></tr>
<tr><td>^ ^=</td><td>Różnica symetryczna (XOR)</td><td>a = b ^ c</td></tr>
<tr><td>~ ~=</td><td>Uzupełnienie do 1 (NOT)</td><td>a = ~b</td></tr>
<tr><td>>> >>=</td><td>Przesunięcie w prawo</td><td>a = b >> 1</td></tr>
<tr><td><< <<=</td><td>Przesunięcie w lewo</td><td>a = b << 1</td></tr>
</table>

Najpierw jednak trzeba sobie przypomnieć czym się różni postać binarna liczby od dziesiętnej. Najlepiej pokazać to na przykładzie.

| **Liczba dziesiętna** | **Liczba binarna** |
|---|---|
| 42 | 101010 |
| 15 | 001111 |

Omówimy sobie teraz kolejne operatory i to jak one działają.

**operator & czyli iloczyn bitowy lub bitowe AND.**

Bitowe AND to prosta czynność sprawdzenia układu zer i jedynek w liczbach poddawanych operacji. Tylko gdy obie liczby wejściowe są równe jeden, to wynik też jest równy jeden.

| **&** | **A** | **B** |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

Zobaczmy to na przykładzie.

| **Liczba dziesiętna** | **Liczba binarna** | | | | | |
|---|---|---|---|---|---|---|
| 42 | 1 | 0 | 1 | 0 | 1 | 0 |
| 15 | 0 | 0 | 1 | 1 | 1 | 1 |
| **10** | **0** | **0** | **1** | **0** | **1** | **0** |

Wynikiem jest liczba 10.

**operator | czyli suma bitowa lub bitowe OR.**

Bitowe OR to prosta czynność sprawdzenia układu zer i jedynek w liczbach poddawanych operacji. Jeśli choć jedna z liczb wejściowych wynosi jeden, to wynik też jest równy jeden.

<table>
<tr><th>|</th><th>A</th><th>B</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

Zobaczmy to na przykładzie.

| **Liczba dziesiętna** | **Liczba binarna** | | | | | |
|---|---|---|---|---|---|---|
| 42 | 1 | 0 | 1 | 0 | 1 | 0 |
| 15 | 0 | 0 | 1 | 1 | 1 | 1 |
| **47** | **1** | **0** | **1** | **1** | **1** | **1** |

Wynikiem jest liczba 47.

**operator ^ czyli symetryczna różnica bitowa lub bitowe XOR.**

Bitowe XOR to sprawdzenie czy zera i jedynki są różnowartościowe w liczbach poddawanych operacji. Gdy jedna z nich jest równa zero, a druga jedne (lub na odwrót), to wynik też jest równy jeden.

| **^** | **A** | **B** |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

Zobaczmy to na przykładzie.

| **Liczba dziesiętna** | **Liczba binarna** | | | | | |
|---|---|---|---|---|---|---|
| 42 | 1 | 0 | 1 | 0 | 1 | 0 |
| 15 | 0 | 0 | 1 | 1 | 1 | 1 |
| **37** | **1** | **0** | **0** | **1** | **0** | **1** |

Wynikiem jest liczba 37.

a ^ b ^ b= a

**operator ~ czyli negacja bitowa lub bitowe NOT.**

Ten operator zamienia bity: zero na jeden, a jeden na zero.

| **Liczba dziesiętna** | **Liczba binarna** | | | | | |
|---|---|---|---|---|---|---|
| 15 | 0 | 0 | 1 | 1 | 1 | 1 |
| **48** | **1** | **1** | **0** | **0** | **0** | **0** |

Wynikiem jest 21.

**operator << służy do przesuwania bitów w lewo.**

Operator ten przesuwa wszystkie bity o jedna pozycję w lewo. Na ostatnie (najbardziej położone na prawo) pozycję wstawiamy 0.

a<<1 – przesuwamy bity o jedną pozycję w lewo
a<<2 – przesuwamy bity o dwie pozycje w lewo
a<<3 – przesuwamy bity o trzy pozycje w lewo itd...

Jak wygląda to w praktyce?

| **Liczba dziesiętna** | **Liczba binarna** | | | | | | |
|---|---|---|---|---|---|---|---|
| 42 | ← | 1 | 0 | 1 | 0 | 1 | 0 |
| **84** | **1** | **0** | **1** | **0** | **1** | **0** | **0** |

Zauważyć można, że liczba, którą uzyskaliśmy jest 2 razy większa od wejściowej.

Poruszając się w lewo o jedno miejsce wartość zwiększa się dwukrotnie. Skoro wszystkie bity przesunięto o jedno miejsce w lewo to nastąpiło jakby pomnożenie całej liczby przez 2.

A co się stanie jeśli przesuniemy liczbę o 2 miejsca w lewo?

| **Liczba dziesiętna** | **Liczba binarna** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 42 | ← | | 1 | 0 | 1 | 0 | 1 | 0 |
| **168** | **1** | **0** | **1** | **0** | **1** | **0** | **0** | **0** |

Przy przesunięciu o 2 miejsca w lewo liczba powiększa się 4 razy. Łatwo więc dojść, że przesunięcie o **n** miejsc odpowiada pomnożenie przez $\mathbf{2^n}$.

Przesunięcia bitowe są bardzo przydatne. Mogą zostać użyte zamiast mnożenia, które jest bardzo powolne w stosunku do przesunięć.

**operator >> służy do przesuwania bitów w prawo.**

Jak łatwo się domyśleć ten operator działa odwrotnie do swego poprzednika. Oznacza to, że można stosować go do dzielenia liczb.

Jednak to dzielenie może być wykonane tylko na liczbach całkowitych.

| **Liczba dziesiętna** | **Liczba binarna** | | | | | | |
|---|---|---|---|---|---|---|---|
| 42 | 1 | 0 | 1 | 0 | 1 | 0 | → |
| **21** | **0** | **1** | **0** | **1** | **0** | **1** | |

Bit położony najbardziej w lewo zostanie uzupełniony cyfrą zero.

Jak wygląda sprawa przy przesunięciu o więcej pozycji? Np. o dwa miejsca?

| **Liczba dziesiętna** | **Liczba binarna** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 42 | 1 | 0 | 1 | 0 | 1 | 0 | | → |
| **10** | **0** | **0** | **1** | **0** | **1** | **0** | | |

Również następuje dzielenie. Ale widzimy, ze program jednocześnie zaokrągla to do liczby całkowitej (w dół). Przy większej ilości kroków wynikiem będą same zera.

http://www.algorytm.edu.pl/funkcje/69-zamiana-liczby-dziesietnej-na-binarna.html

http://guidecpp.cal.pl/cplus,operators-bits,

http://www.algorytm.org/kurs-algorytmiki/operacje-bitowe.html

http://www.algorytm.edu.pl/wstp-do-c/operatory-w-c.html

http://eduinf.waw.pl/inf/alg/002_struct/0007.php

https://pl.wikibooks.org/wiki/C/Operatory#Inne_operatory

# Dodatek A – zestawienie podstawowych funkcji C/C++

**Tabela 27 -Słowa kluczowe w języku C++**

| | | | | | |
|---|---|---|---|---|---|
| *asm* | default | goto | register | *template* | volatile |
| auto | do | if | restrict | *this* | while |
| break | double | inline | return | *throw* | _Bool |
| case | else | int | short | *try* | _Complex |
| *catch* | enum | long | signed | typedef | _Imaginary |
| char | extern | *new* | sizeof | union | |
| *class* | float | *operator* | static | unsigned | |
| const | for | *private* | struct | virtual | |
| continue | *friend* | *protected* | switch | void | |

**Tabela 28 - Zestawienie operatorów języka C/C++**

| Symbol | Nazwa operatora | Przykład |
|---|---|---|
| - | Zmiana znaku | x= -5; |
| + | Tożsamość znaku (bez zmiany) | x= +5; |
| - | Odejmowanie | y=x-5; |
| + | Dodawanie | y=x+5; |
| * | Mnożenie | y=x*3; |
| \ | Dzielenie | y=x\12; |
| % | Reszta z dzielenia<br>(dzielenie modulo) | y=x%7; |
| x+=y | Dodaj | x+=5; |
| x-=y | Odejmij | x-=5; |
| x*=y | Pomnóż przez | x*=3; |
| x/=y | Podziel przez | x/=12; |
| x%=y | Podziel modulo przez | x%=7; |
| -- | *Dekrementacja* (zmniejszenie o 1) | |
| | Predekrementacja | --x; |
| | Posdekrementacja | x--; |
| ++ | *Inkrementacja* (zwiększenie o 1) | |
| | Preinkrementacja | ++x; |
| | Postinkrementacja | x++; |
| = | Podstawienie | x=5; |
| > | Większy | x>y |
| >= | Większy lub równy | x>=y |
| < | Mniejszy | z<t |
| <= | Mniejszy lub równy | z<=t |
| == | Równy | x==5 |
| != | Różny od | x!=7 |
| && | Koniunkcja (AND) | c = a && b; |
| \|\| | Alternatywa (OR) | c = a \|\| b; |

<table>
<tr><td>!</td><td>Negacja (NOT)</td><td>c = !a;</td></tr>
<tr><td>&</td><td>Iloczyn bitowy (AND)</td><td></td></tr>
<tr><td>|</td><td>Suma bitowa (OR)</td><td></td></tr>
<tr><td>^</td><td>Różnica symetryczna (XOR)</td><td></td></tr>
<tr><td>~</td><td>Uzupełnienie do 1 (NOT)</td><td></td></tr>
<tr><td>>></td><td>Przesunięcie w prawo</td><td></td></tr>
<tr><td><<</td><td>Przesunięcie w lewo</td><td></td></tr>
<tr><td>?</td><td>Operator warunkowy</td><td>x<0 ? x--: x++;</td></tr>
<tr><td>&</td><td>Adres miejsca pamięci</td><td>m=&licznik;</td></tr>
<tr><td>*</td><td>Wartość zmiennej w danym miejscu pamięci</td><td>x=*m;</td></tr>
<tr><td>sizeof</td><td>Operator rozmiaru zmiennej</td><td>cout<<x<<sizeof x;</td></tr>
<tr><td>,</td><td>Operator wiązania</td><td>x=(y=y-5, 25/y);</td></tr>
<tr><td>.</td><td>Dostęp do struktury i unii</td><td>x.a=12;</td></tr>
<tr><td>-></td><td>Dostęp do wskaźnika do struktury i unii</td><td>x1->a=12;</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
</table>

**Tabela 29 - Zestawienie funkcji matematycznych w C/C++**

| | | |
|---|---|---|
| sin(x) | sinus (x) | `z = sin(1.0);` |
| cos(x) | cosinus (x) | `z = cos(1.0);` |
| tan(x) | tangens (x) | `z = tan(1.0);` |
| asin(x) | arcus sinus (x) | `z = asin(-0.5);` |
| acos(x) | arcus cosinus (x) | `z = acos(-0.5);` |
| atan(x) | arcus tangens(x) | `z = atan(-0.5);` |
| atan2(x,y) | arcus tangens(x/y) | `z = atan2(2,3);` |
| sinh(x) | sinus hiperboliczny (x) | `sh = sinh (0.7);` |
| cosh(x) | cosinus hiperboliczny (x) | `ch = cosh (x);` |
| tanh(x) | tangens hiperboliczny (x) | `th = tanh (-0.1);` |
| log(x) | logarytm naturalny x | `cout<<log(2.71);` |
| log10(x) | logarytm dziesiętny x | `cout<<log10(100);` |
| pow(a,b); | Oblicza $a^b$ | `z = pow(10,2);` |
| pow10(b); | Oblicza $10^b$ | `z = pow10(2);` |
| exp(x) | Eksponenta x ($e^x$) | `cout<<exp(1.0);` |
| ldexp(m,w) | Oblicza wynik $m*2^w$ | `x=ldexp(mant,wykl);` |
| ceil(x) | Najmniejsza liczba całkowita większa od x | `ceil(3.7);` |
| floor(x) | Największa liczba całkowita mniejsza od x | `floor(3.7);` |
| fmod(x,y) | Reszta z dzielenia x przez y | `fmod(13,5);` |
| hypot(a,b) | Oblicza w trójkącie długość przeciwprostokątnej | `hypot(4,3);` |
| sqrt(x) | Pierwiastek kwadratowy z x | `cout<<sqrt(4);` |
| fabs(x) | Wartość bezwzględna z x | `fabs(-2.7);` |