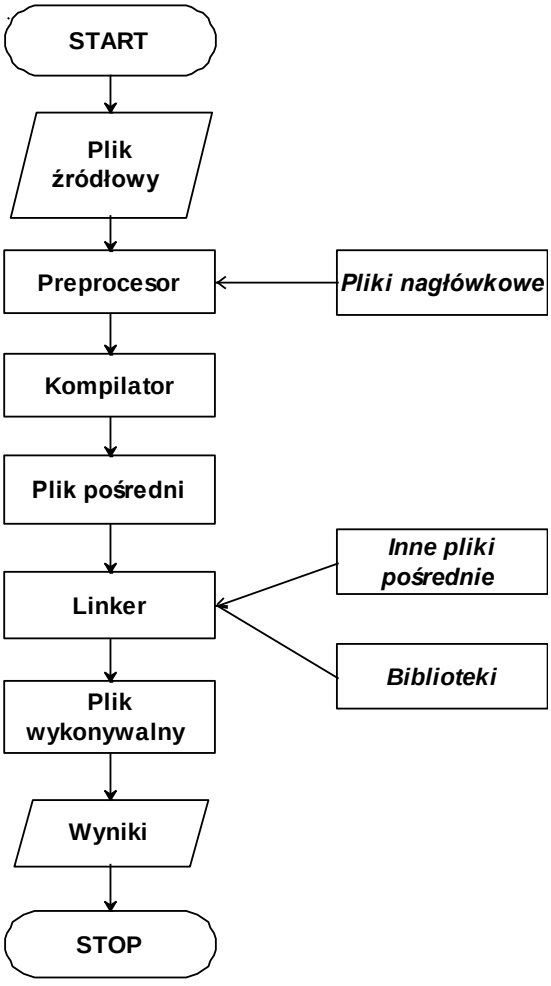
Wprowadzenie do języka C++

Język C++ jest językiem realizującym kilka paradygmatów (wzorców) programowania, przez co ma olbrzymie możliwości i powstaje w nim bardzo dużo oprogramowania na komputery osobiste. Ponadto, ucząc się języka C++, łatwo później przejść na inny język programowania. Do początkowego posługiwania się tym językiem i pisania całkiem użytecznych programów wystarczy znajomość kilkunastu tzw. słów kluczowych oraz gramatyki (składni) języka C++. Jeśli dodać do tego pogłębione na wcześniejszych lekcjach algorytmiki logiczne myślenie i kreatywność, naprawdę bardzo szybko można tworzyć użyteczne programy, z czasem poznając coraz więcej słów kluczowych i możliwości tego języka.

Język C++ zalicza się do języków wysokiego poziomu. Oznacza to, że programista pisze programy w języku podobnym do języka ludzkiego (w tym wypadku angielskiego). Jest to tzw. kod (program) źródłowy, zapisywany w pliku (z rozszerzeniem `.cpp`). Natomiast kompilator tłumaczy kod źródłowy na język zrozumiały dla komputera (tzw. kod maszynowy), w wyniku czego powstaje plik wykonywalny (z rozszerzeniem `.exe`).

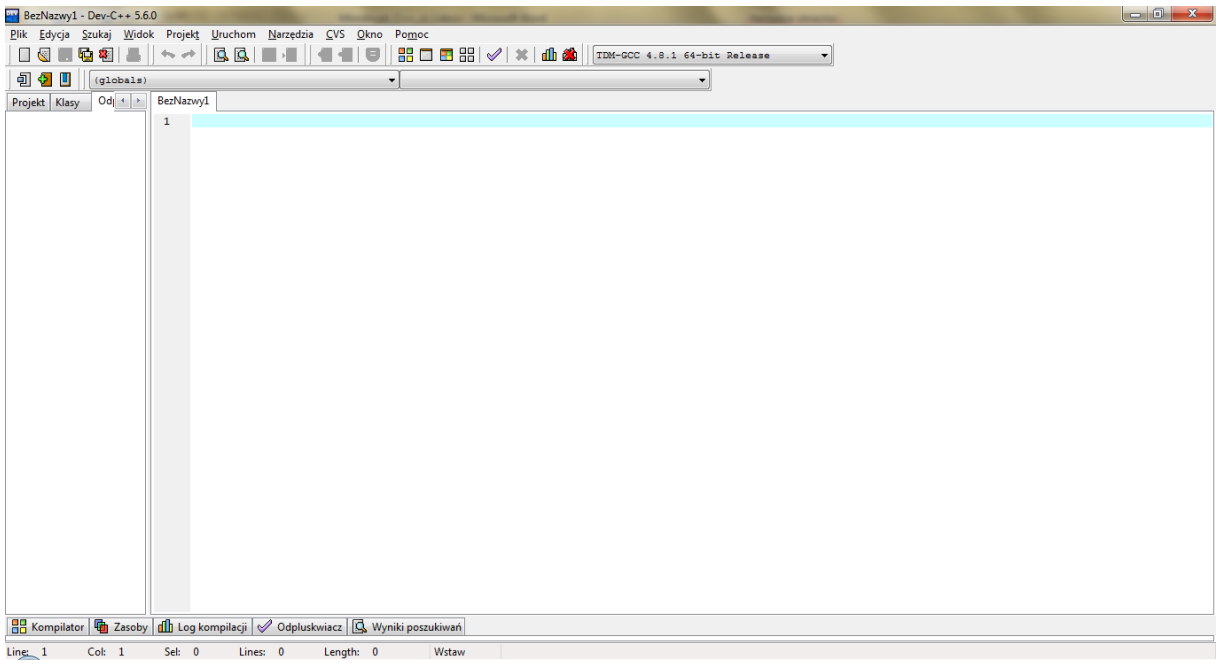
Po drodze preprocesor (integralna część kompilatora) wstępnie przetwarza kod, uwzględniając dołączane przez programistę tzw. pliki nagłówkowe (z rozszerzeniem `.h` lub `.hpp`). Następnie kompilator tworzy plik pośredni (z rozszerzeniem `.o` lub `.obj`), zaś linker (konsolidator), który również stanowi część składową kompilatora, dodaje do niego w razie potrzeby inne pliki pośrednie oraz pliki biblioteczne (z rozszerzeniem `.a`, `.lib` lub `.so`).

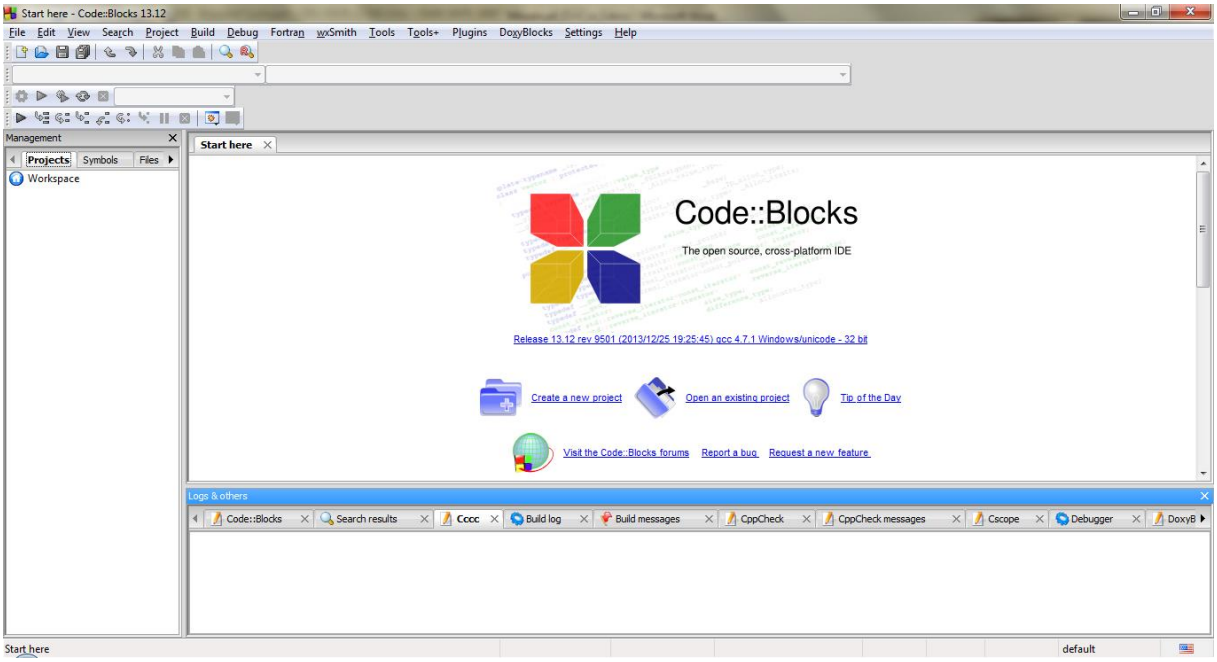


Pisanie i uruchamianie programów komputerowych składa się zatem z kilku faz i niezbędne staje się posiadanie co najmniej odpowiedniego programu tłumaczącego (kompilatora), a jeszcze lepiej zintegrowanego środowiska programistycznego, oznaczanego w skrócie jako IDE (ang. *Integrated Development Environment*).

Na poziomie amatorskim i półprofesjonalnym wymagania te najlepiej spełniają obecnie dwa bardzo popularne i darmowe IDE, tj. Dev-C++ oraz Code::Blocks, z tym że drugie ma taką przewagę nad pierwszym, iż posiada

swoje wersje nie tylko dla MS Windows, lecz także dla Linux i MacOS. Dev-C++ wyposażony jest dla odmiany w polski interfejs użytkownika, co wcale nie jest takie dobre, ponieważ przyszły programista od początku powinien posługiwać się językiem angielskim. Nie ma jednak problemu, by w opcjach środowiska Dev-C++ (w menu Narzędzia) przełączyć się na inny język.





Należy pamiętać, aby pobierać programy z oryginalnych stron internetowych producentów oprogramowania, gdyż w ten sposób możemy uniknąć przykrych niespodzianek w postaci niepotrzebnych dodatków reklamowych czy nawet wirusów komputerowych. Dla Dev-C++ jest to strona http://www.bloodshed.net/devcpp.html, natomiast Code::Blocks można ściągnąć ze strony http://www.codeblocks.org/ – zawsze poszukując aktualnej i stabilnej wersji w zakładkach typu Downloads. W Internecie można też znaleźć wersje portable (przenośne) obu IDE, a zatem nie wymagające instalowania, przez co możliwe jest także tworzenie programów np. z poziomu pen-drive'a na dowolnym komputerze, do którego można podłączyć napęd przenośny. To dobry pomysł, ale trzeba uważać na potencjalnie złośliwe dodatki. Instalacja oraz uruchomienie obu środowisk jest intuicyjne i proste (w trakcie instalacji należy wybrać opcje dające największe możliwości, aby uniknąć przykrych niespodzianek w przyszłości).

W razie problemów można zwrócić się do nauczycieli przedmiotów informatycznych, ponieważ głównym zadaniem tego miniskryptu jest nauka pisania programów. Dlatego nie ma w nim szczegółowych zrzutów ekranowych, obrazujących poszczególne fazy tworzenia i kompilowania programów, a opis funkcji wykorzystywanych w programach jest maksymalnie skrócony. W razie potrzeby można skorzystać z wyszukiwarki lub skorzystać z linków umieszczonych w miniskrypcie, które odsyłają do opisów wyczerpujących temat. Dodatkowe wyjaśnienia pojawią się też na lekcjach programowania.

Wszystkie programy z miniskryptu przetestowane zostały w środowisku Code::Blocks w wersji 16.01 pracującym pod systemem operacyjnym MS Windows 7.

Najprostsze programy w języku C++

Teoretycznie najprostszy program, jaki można utworzyć i skompilować w C++, ma postać:

```
/*Pierwszy
program*/

main()//funkcja główna
{
}
```

Uwagi:

1. Program wykonuje się linia po linii, od lewej do prawej strony.
2. W każdym programie napisanym w języku C++ musi znaleźć się funkcja `main()`, zwana funkcją główną, niekoniecznie jednak musi być umieszczona na początku programu. Do niej kieruje się komputer po uruchomieniu programu i to ona steruje jego wykonaniem.
3. Wielkość liter ma znaczenie w języku C++, czyli `Main()` i `main()` są to dwie różne funkcje. Instrukcje i funkcje należy pisać małymi literami alfabetu, przy czym nazwy funkcji nie mogą zawierać polskich znaków diakrytycznych (tzw. ogonków).
4. Ciało funkcji (zestaw instrukcji do wykonania) znajduje się między nawiasem klamrowym otwierającym `{` i zamykającym `}`. W tym przykładzie funkcja `main()` nie wykonuje żadnych dodatkowych czynności.
5. Kompilator może wygenerować dla powyższego programu ostrzeżenie, bo zgodnie ze standardem języka C++ funkcja `main()` musi mieć zadeklarowany typ, na przykład `int main()`. Skrót `int` pochodzi od słowa *integer*, czyli liczba całkowita (niepodzielna). Jednak mimo braku `int` program da się skompilować i będzie działał poprawnie. O typach danych będzie nieco później. W starszych programach można też spotkać przed funkcją (a także w nawiasach klamrowych) słowo `void` (ang. *void* – pusty, próżny), które oznacza, że funkcja nie zwraca żadnej wartości lub nie potrzebuje żadnych parametrów w nawiasach okrągłych.
6. Para dwuznaków `/*` oraz `*/` służy do umieszczenia pomiędzy nimi nieobowiązkowych, ale bardzo zalecanych komentarzy programisty, które jednak nie powinny być zbyt banalne (tutaj umieszczono je wyłącznie ze względów dydaktycznych). Komentarze tego typu mogą zajmować więcej niż jeden wiersz programu i służą programistom np. do tego, aby nazwać program lub jego fragment albo objaśnić trudniejsze fragmenty kodu. Są one całkowicie pomijane przez kompilator. Dzięki temu komentarze można wykorzystywać również do „wyłączenia" jakiegoś fragmentu programu np. w poszukiwaniu błędu. Innym przykładem komentarza (nieco prostszego) jest para znaków `//`, po których umieszcza się objaśnienie, ale zajmujące miejsce tylko do końca wiersza. Wyjście z komentarzem poza wiersz zawierający `//` spowoduje wygenerowanie błędu.

Łatwo zauważyć, że za każdym uruchomieniem nowego projektu środowisko Code::Blocks otwiera programiście plik o nazwie `main.cpp`, w którym znajduje się już gotowy zestaw instrukcji do wykonania. Nb. wiele środowisk programistycznych zaczyna naukę programowania od wyświetlenia na ekranie klasycznego tekstu *Hello world!*.

```
#include <iostream>
using namespace std;
```

```
int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Jak widać, program ten jest nieco bardziej skomplikowany. Zauważ, że każda instrukcja programu zakończona jest średnikiem `;`. Znaczenie poszczególnych linii kodu jest następujące:

`#include <iostream>` – żądanie dołączenia (ang. *include*) standardowej biblioteki o nazwie `iostream`, której zadaniem jest obsługa strumieni (ang. *stream*) wejścia/wyjścia (ang. *input/output*). Dołączenie żądanej biblioteki jest konieczne ze względu m.in. na późniejsze wykorzystanie opisanego w niej operatora strumienia wyjścia `cout` (czytaj: *si-aut*), który zajmuje się wyświetleniem napisu *Hello world!* na ekranie. Deklaracje tego typu zawsze znajdują się na początku programu. Deklaracja ta nazywana jest także dyrektywą preprocesora, ponieważ wykonywana jest przez specjalny program (preprocesor) uruchamiany przed właściwym kompilatorem. Zadaniem preprocesora jest wstępne przygotowanie kodu programu w taki sposób, aby zawierał wszystko to, co potrzebne jest kompilatorowi do przetłumaczenia całego programu na kod maszynowy. Nazwa biblioteki może być ujęta w nawiasy ostre `<nazwa>` lub w cudzysłowy `"nazwa"`, przy czym `nazwa` biblioteki może też zawierać rozszerzenie `.h`. Różnica polega na tym, że w wypadku zapisu `<nazwa>` kompilator szuka danej biblioteki tylko w katalogu domyślnym, a w wypadku zastosowania napisu `"nazwa"` można podać ścieżkę dostępu do katalogu zawierającego odpowiedni plik.

`using namespace std;` – jest to deklaracja użycia tzw. *standardowej przestrzeni nazw*, które pomagają programiście w ten sposób, że umożliwiają np. tworzenie bez kolizji funkcji o takich samych nazwach i deklaracjach w różnych przestrzeniach nazw, ale wykonujących inne czynności. Pełnią więc funkcję podobną do katalogów, ponieważ w różnych katalogach mogą znajdować się pliki o takich samych nazwach, ale działające zupełnie inaczej. Tutaj taką rolę pełni gotowa przestrzeń `std`, która zawiera bardzo dużo przydatnych narzędzi (np. do wyszukiwania i sortowania), a dzięki takiej deklaracji kompilator jest poinformowany, gdzie ma szukać tych narzędzi, czyli uzyskuje niejako „ścieżkę dostępu" do katalogu. Programista może także definiować własne przestrzenie nazw. W tym przykładzie, gdyby nie było tej deklaracji, linia `cout << "Hello world!" << endl;` (i wszystkie inne tego typu) musiałaby być zmodyfikowana do postaci `std::cout << "Hello world!" << std::endl;`. Podwójny dwukropek `::` nazywany jest operatorem zasięgu i informuje kompilator, że operacja będzie wykonywana na zmiennej globalnej, czyli dostępnej w całym programie, w odróżnieniu od zmiennych lokalnych, które mają zasięg jedynie w określonej części programu.

`cout << "Hello world!" << endl;` – zawiera polecenie wyświetlenia napisu ujętego w cudzysłów oraz przejście do nowej linii, realizowane za pomocą manipulatora `endl` (`endl` jest skrótem od `end of line`). Ten sam efekt można uzyskać za pomocą instrukcji `cout << "Hello world!\n";`. Tutaj dwuznak `\n` oznacza tak samo przejście do nowej linii. Różnica między nimi jest taka, że manipulator `endl` zawiera w sobie dwuznak `\n` oraz dodatkowo czyści bufor pamięci. Zatem `\n` jest szybsze, ale nie czyści bufora. W praktyce, mimo że efekt działania obu manipulatorów jest identyczny, częściej używa się `endl`. Zwróć też uwagę, że operator przepływu strumienia danych, oznaczony za pomocą dwuznaku `<<`, skierowany jest w stronę operatora strumienia wyjścia `cout` (ang. ***c**onsole **out**put*).

`return 0;` – polecenie `return` służy do zwracania przez różne funkcje jakiejś wartości lub kodu błędu. Jeśli zostanie pominięte, funkcja zwróci wartość zero. Jest to ostatnia linia programu i poza nią nie umieszcza się innych linii programu należących do funkcji `main()`, bo się nie wykonają. W tym wypadku, ponieważ funkcja `int main()` i tak nie miała niczego zwracać (poza oddaniem sterowania do systemu operacyjnego), zwróciłaby wartość zero (jeśli program zadziała bezbłędnie). Sprawdź to wygaszając linię znakiem `//` umieszczonym przed poleceniem `return 0`. Jak widać, w obu przypadkach komunikat w oknie Code::Blocks po zakończeniu programu jest taki sam. Byłoby więc można pominąć polecenie `return 0`, jednak z uwagi na zgodność ze standardem języka C++, nie należy tego czynić. Sprawdź też, co się stanie, jeśli zmodyfikujesz polecenie np. do postaci `return -1`. Program da się poprawnie wykonać, ale kompilator Code::Blocks zauważy różnicę i wygeneruje podświetlony na czerwono komunikat.

Uwagi:

1. Po skompilowaniu programu i uruchomieniu go w konsoli MS Windows, nie da się zauważyć jego działania, ponieważ okno programu błyskawicznie znika. Można temu zaradzić modyfikując program na kilka sposobów, które będą przedstawione później.
2. Opis standardowych bibliotek języka C++ oraz bardzo dużo innych pożytecznych informacji można znaleźć pod adresem http://www.cplusplus.com/ w dziale *Reference*.

Proste programy sekwencyjne

W języku C++ istnieje możliwość wykonywania różnorodnych operacji matematycznych, które możliwe są dzięki podstawowymi symbolom:

Operator	Opis	Przykład użycia
`=`	przypisanie (lewa strona przyjmuje wartość prawej)	`a = b`
`+`	dodawanie	`a + b`
`-`	odejmowanie	`a - b`
`*`	mnożenie	`a * b`
`/`	dzielenie (gdy obie zmienne są całkowite, to wynikiem jest część całkowita z dzielenia)	`a / b`
`%`	modulo (reszta z dzielenia liczby całkowitej przez inną liczbę całkowitą)	`a % b`
`++`	inkrementacja (zwiększenie wartości zmiennej o 1)	`a++`
`--`	dekrementacja (zmniejszenie wartości zmiennej o 1)	`a--`

Uwaga:
Inkrementacja jest odpowiednikiem operacji przypisania `a = a + 1`, natomiast dekrementacja `a = a - 1`. Zarówno inkrementacja, jak i dekrementacja występuje w języku C++ w dwóch odmianach pod postacią:

1. **Postinkrementacji**, oznaczanej jako `a++`, która zwiększa wartość `a` po użyciu tej zmiennej.
2. **Preinkrementacji**, oznaczanej jako `++a`, która zwiększa wartość `a` przed użyciem tej zmiennej.
3. **Postdekrementacji**, oznaczanej jako `a--`, która zmniejsza wartość `a` po użyciu tej zmiennej.
4. **Predekrementacji**, oznaczanej jako `--a`, która zmniejsza wartość `a` przed użyciem tej zmiennej.

Do pisania programów obliczeniowych język C++ oferuje m.in. proste typy danych, które zostały zgromadzone w poniższej tabeli.

Nazwa typu	Typ	Wielkość	Zakres
`void`	bezwartościowy	0 B	nie zwraca żadnej wartości
`bool`	logiczny	1 B	prawda/fałsz
`char`	całkowity	1 B	-128..127
`unsigned char`	całkowity	1 B	0..255
`short int`	całkowity	2 B	$-2^{15}...2^{15}-1$
`unsigned short int`	całkowity	2 B	$0..2^{16}-1$
`int`	całkowity	4 B	$-2^{31}...2^{31}-1$
`unsigned int`	całkowity	4 B	$0..2^{32}-1$
`long long int`	całkowity	8 B	$-2^{63}...2^{63}-1$
`unsigned long long int`	całkowity	8 B	$0..2^{64}-1$
`float`	zmiennoprzecinkowy	4 B	$\pm 3.4 \cdot 10^{38}$
`double`	zmiennoprzecinkowy	8 B	$\pm 1.7 \cdot 10^{308}$
`long double`	zmiennoprzecinkowy	12 B	$\pm 1.1 \cdot 10^{4932}$

Uwaga:
W typach zmiennoprzecinkowych (ang. *floating point*), które są odpowiednikiem liczb rzeczywistych w matematyce, istotna jest również informacja, z jaką precyzją (dokładnością) zapamiętywana jest dana liczba. Dla `float` jest to 7-8 cyfr znaczących, dla `double` 15-16, a dla `long double` 19-20 cyfr znaczących. Jeśli nie ma specjalnych ograniczeń pamięciowych, częstym wyborem programisty w stosowaniu liczb zmiennoprzecin-

kowych jest typ `double`. Oszczędnościowy typ `float` pochodzi bowiem z czasów, gdy pamięć komputerów – ze względu na bardzo wysoką cenę – była niewielka. W miniskrypcie, ze względów dydaktycznych, typy zmiennoprzecinkowe stosowane są zamiennie.

Przykład
Napisz program obliczający pole prostokąta P o bokach a i b.

```
#include <iostream>
using namespace std;
int main()
{
    double a, b, P;
    cout << "Podaj dlugosc boku a: ";
    cin >> a;
    cout << "Podaj dlugosc boku b: ";
    cin >> b;
    P = a * b;
    cout << "Pole prostokata wynosi: " << P << endl;
    return 0;
}
```

Uwagi:

1. Program ilustruje użycie zmiennych, które mogą zmieniać swoją wartość przy każdym uruchomieniu programu. Nazwy zmiennych powinny odzwierciedlać jak najlepiej reprezentowane przez siebie wartości. W nazwach zmiennych nie wolno stosować polskich znaków diakrytycznych (tzw. ogonków) oraz tzw. słów kluczowych (ang. *keywords*) języka C++, do których zaliczają się m.in.: `asm`, `auto`, `break`, `case`, `catch`, `char`, `class`, `const`, `continue`, `default`, `delete`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `friend`, `goto`, `if`, `inline`, `int`, `long`, `new`, `operator`, `private`, `protected`, `public`, `register`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `template`, `this`, `throw`, `try`, `typedef`, `union`, `unsigned`, `virtual`, `void`, `volatile`, `while`.
2. Operator strumienia wejściowego `cin` (czytaj: *si-in*) odpowiada za wczytanie informacji z klawiatury (ang. ***c**onsole **in**put*).

Przykład
Napisz program obliczający pole dowolnego trójkąta, jeśli dane są jedynie długości jego boków.

Wskazówka: należy zastosować wzór Herona: $S = \sqrt{p(p-a)(p-b)(p-c)}$, gdzie S – pole trójkąta; a, b i c – długości boków, p – połowa obwodu trójkąta. Aby wyznaczyć pierwiastek kwadratowy, w języku C++ należy użyć funkcji `sqrt()`, gdzie w nawiasy należy wpisać argument funkcji, na przykład $\sqrt{2}$ zapisujemy jako `sqrt(2)`. Ponadto należy zadeklarować użycie biblioteki zawierającej funkcje matematyczne, która nosi nazwę `<math.h>` albo `<cmath>`, która jest jej nowszą odmianą i jest w pełni zgodna ze standardem języka.

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

using namespace std;

int main()
{
    double a, b, c, p, S;
    cout << "Podaj bok a: ";
    cin >> a;
    cout << "Podaj bok b: ";
    cin >> b;
    cout << "Podaj bok c: ";
    cin >> c;
    p = (a + b + c)/2;
```

```
    S=sqrt(p*(p-a)*(p-b)*(p-c));
    cout << "Pole trojkata wynosi: " << S << endl;
    return 0;
}
```

Przykład
Napisz program, który pyta użytkownika o imię i wiek, a następnie wyświetla odpowiedni komunikat na ekranie wykorzystujący wprowadzone dane.

```
#include <iostream>

using namespace std;

string imie; //typ string zarezerwowany jest dla łańcuchów znakowych
int wiek;

int main()
{
  cout << "Jak masz na imie?" << endl;
  cin >> imie;
  cout << "Ile masz lat?" << endl;
  cin >> wiek;
  cout << "Masz na imie " << imie << " i masz " << wiek << " lat." << endl;
  return 0;
}
```

Przykład
Wykorzystując funkcję `sizeof()`, która zwraca liczbę bajtów zajmowanych przez daną zmienną lub typ danych, napisz program ilustrujący wykorzystanie pamięci operacyjnej przez różne typy danych.

```
#include <iostream>

using namespace std;

int main()
{
 cout << "                    int : " << sizeof (int) << endl
      << "           unsigned int : " << sizeof (unsigned) << endl
      << "              short int : " << sizeof (short) << endl
      << "     unsigned short int : " << sizeof (unsigned short) << endl
      << "                   char : " << sizeof (char) << endl
      << "          unsigned char : " << sizeof (unsigned char) << endl
      << "          long long int : " << sizeof (long long) << endl
      << "unsigned long long int : " << sizeof (unsigned long long) << endl
      << "                  float : " << sizeof (float) << endl
      << "                 double : " << sizeof (double) << endl
      << "            long double : " << sizeof (long double) << endl
      << "                   bool : " << sizeof (bool) << endl;

 return 0;
}
```

Uwaga:
Zwróć uwagę, że w programie występuje tylko jedna instrukcja `cout`, zakończona średnikiem, która zapisana jest w wielu wierszach programu.

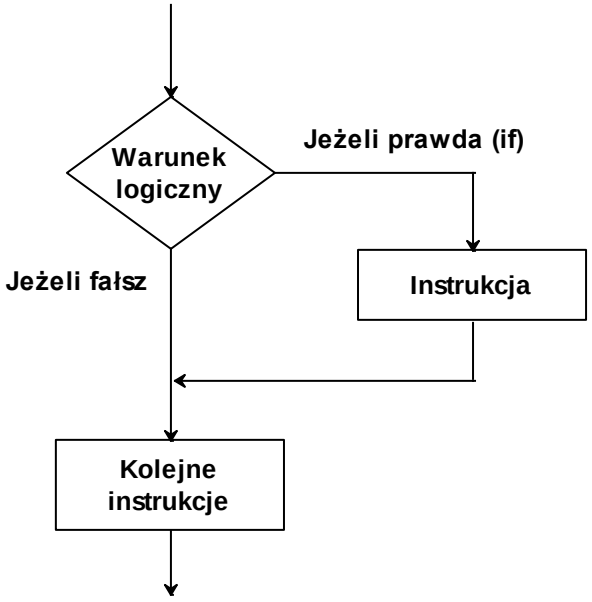
Przykład
Napisz program ilustrujący działanie zmiennej lokalnej i zmiennej globalnej.

```
#include <iostream>

using namespace std;

int z=0; //deklaracja zmiennej globalnej z
int main()
{
 int y ; //deklaracja zmiennej lokalnej y
 double z; //deklaracja zmiennej lokalnej z
 z = 7.258; //przypisanie wartości do zmiennej lokalnej z
 ::z = 2; //przypisanie wartości do zmiennej globalnej z
 y = ::z; //przypisanie wartości zmiennej globalnej do zmiennej lokalnej
 cout << "Wartosc zmiennej lokalnej z wynosi: " << z << endl;
 cout << "Wartosc zmiennej globalnej z wynosi: " << y << endl;

 return 0;
}
```

Uwaga:
W przykładzie deklaracja zmiennej globalnej `z` czyni ją widzialną dla całego programu. Zasięg zmiennej `z` typu `int` obejmuje zatem cały plik z programem źródłowym. Tak określona zmienna `z` może być używana w bloku funkcji `main()`, aż do napotkania deklaracji zmiennej lokalnej `z`, o tej samej nazwie, ale typu `double`, która przesłoniła (ukryła) wcześniejszą zmienną globalną. Mimo to dostęp do zmiennej globalnej nie został bezpowrotnie stracony, a to dzięki operatorowi zasięgu o symbolu „`::`”. Jak widać, identyfikator zmiennej `z`, poprzedzony operatorem zasięgu `::`, zapewnia dostęp do zmiennej globalnej.

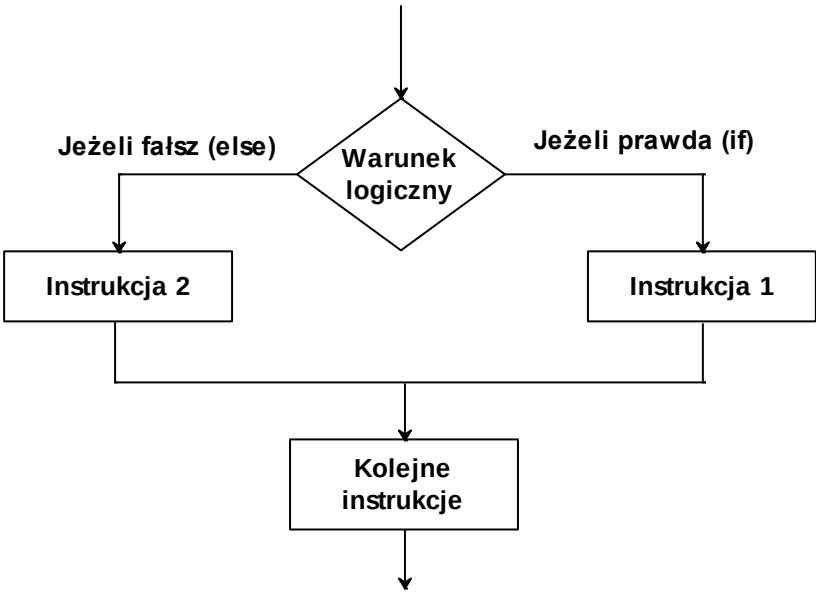
Programy z instrukcją warunkową `if – else`

Instrukcja warunkowa (decyzyjna) `if – else` występuje w dwóch odmianach, tj. uproszczonej i pełnej. Postać uproszczona, przedstawiona na rysunku poniżej, polega na sprawdzeniu warunku logicznego za pomocą instrukcji `if`. Jeśli warunek logiczny jest prawdziwy, wówczas wykonywana jest `Instrukcja`, a po niej kolejne instrukcje programu. Jeśli warunek logiczny nie jest prawdziwy, od razu następuje przejście do kolejnych instrukcji programu.



W postaci pełnej, przedstawionej na rysunku poniżej, jeżeli warunek logiczny sprawdzany za pomocą instrukcji `if` jest prawdziwy, wykonywana jest `Instrukcja 1`, a po niej kolejne instrukcje programu. W przeciwnym

wypadku (`else`) wykonywana jest `Instrukcja 2`, a po niej następuje przejście do kolejnych instrukcji programu.



W języku C++ wartości logicznej `false` (fałsz) odpowiada wartość liczbowa `0`, natomiast wartości logicznej `true` (prawda) odpowiada każda inna liczba. W warunku logicznym instrukcji `if - else` mogą być stosowane następujące operatory (mają one także inne zastosowania w języku C++):

Operator	Opis	Przykład użycia
==	równy	a == b
!=	nierówny (różny)	a != b
>	większy niż	a > b
<	mniejszy niż	a < b
>=	większy lub równy	a >= b
<=	mniejszy lub równy	a <= b
||	suma logiczna dwóch wyrażeń	(a > b) || (b > c)
&&	iloczyn logiczny dwóch wyrażeń	(a > b) && (b > c)
!	negacja	!a

Przykład
Napisz program obliczający iloraz dwóch liczb rzeczywistych $liczba1$ i $liczba2$. Uwzględnij przypadek, gdy $liczba2 = 0$.

```
#include <iostream>
#include <windows.h>
using namespace std;
int main()
{
    float liczba1, liczba2, iloraz;
    cout << "Wprowadz pierwsza liczbe: ";
    cin >> liczba1;
    cout << "Wprowadz druga liczbe: ";
    cin >> liczba2;
    cout << endl;
    if (liczba2 != 0)
    {
        iloraz = liczba1/liczba2;
        cout <<"Iloraz wynosi: " << iloraz;
    }
    else
    {
```

```
        cout << "Blad dzielenia przez zero";
    }
    cout << endl << endl;
    system ("pause");
    return 0;
}
```

Uwaga:
Instrukcja `system ("pause");` powoduje wstrzymanie wykonania programu i wygenerowanie komunikatu: `Aby kontynuować, naciśnij dowolny klawisz...`. Obsługa tej instrukcji znajduje się w bibliotece `<windows.h>`. Jej zastosowanie powoduje, że efekt działania programu nie znika z okna konsoli MS Windows, dzięki czemu użytkownik może się z nim zapoznać. Innym rozwiązaniem wstrzymującym wykonanie programu jest zastosowanie w miejsce `system ("pause");` instrukcji `getchar();`, której obsługa znajduje się w bibliotece `<cstdio>`. Efekt jej działania jest identyczny, ale nie generuje żadnego dodatkowego komunikatu. W niektórych kompilatorach konieczne jest dwukrotne użycie instrukcji `getchar();` z tego powodu, że w rzeczywistości naciśnięcie klawisza `ENTER` składa się z dwóch znaków, oznaczonych heksadecymalnie jako `0Ah` i `0Dh`.

Instrukcje `if – else` mogą być zagnieżdżone, wówczas ich konstrukcja przyjmuje np. postać:

```
if (warunek logiczny 1)
 instrukcja
  else if (warunek logiczny 2)
   instrukcja
    else if (warunek logiczny 3)
    instrukcja
     else
      instrukcja
```

W konstrukcji tej oblicza się kolejno wartości wyrażeń logicznych w nawiasach. Pierwsze prawdziwe napotkane wyrażenie spowoduje wykonanie związanej z nim instrukcji i zakończenie wykonywania całej konstrukcji. Instrukcja może być pojedyncza lub stanowić grupę instrukcji ujętą w nawiasy klamrowe.

Przykład
Napisz program, który wczytuje z klawiatury dowolną liczbę *N* i bada jej znak, tzn. określa, czy jest ona dodatnia, ujemna lub jest zerem.

```
#include <iostream>
using namespace std;
int main()
{
    float N;
    cout << "Podaj liczbe N: ";
    cin >> N;
    cout << endl;
    if (N == 0)
    {
        cout << "Wprowadziles zero" << endl;
    }
     else if (N > 0)
     {
         cout << "Wprowadziles liczbe dodatnia" << endl;
     }
       else
          cout << "Wprowadziles liczbe ujemna" << endl;
    return 0;
}
```

Przy zagnieżdżonym stosowaniu instrukcji `if – else` należy uważać na dwuznaczności, np. w ciągu instrukcji

```
if (n>0)
if (a>b)
z = a;
else
z = b;
```

część związana z `else` dotyczy wyłącznie wewnętrznej (drugiej) instrukcji `if`. Jeśli miałaby dotyczyć pierwszej instrukcji `if`, wówczas należy użyć nawiasów klamrowych.

```
if (n>0)
 {
  if (a>b)
  z = a;
 }
else
 z = b;
```

Błędy związane z niewłaściwym użyciem `else` mogą być bardzo trudne do zlokalizowania!

Czasami warunek logiczny w instrukcji `if – else` może być bardziej złożony, na przykład

```
if ((x <= 3) || (x > 11) || (a != 6) && (a > 12))
```

Analizę prawdziwości tego warunku kompilator prowadzi od lewej do prawej strony. Jeśli dla przykładu `x` jest równy `2`, natomiast `a` jest równe `6`, to wyrażenie `(x <= 3) || (x > 11)` jest prawdziwe, bo pierwszy człon warunku jest prawdziwy, zaś drugi fałszywy, ale oba człony połączone są spójnikiem logicznym OR (LUB). W efekcie daje to prawdę, bo suma logiczna OR daje w wyniku prawdę, jeśli choć jeden warunek jest prawdziwy. Następny człon `(a != 6)` jest fałszywy, ale z poprzednim `(x <= 3) || (x > 11)` (prawdziwym) również połączony jest spójnikiem OR, więc w sumie trzy pierwsze człony dają wartość prawdy logicznej. Ostatni warunek `(a > 12)` jest fałszywy. Ponieważ z poprzednimi trzema członami `((x <= 3) || (x > 11) || (a != 6)`, (które w sumie dały prawdę), połączony jest spójnikiem logicznym AND (I), ostatecznie otrzymamy fałsz, bo iloczyn logiczny daje prawdę wyłącznie wtedy, gdy oba wyrażenia połączone spójnikiem AND są prawdziwe. Można to przedstawić następująco:

```
(x <= 3) || (x > 11) || (a != 6) && (a > 12)

TRUE || (x > 11) || (a != 6) && (a > 12)

TRUE || FALSE || (a != 6) && (a > 12)

TRUE || FALSE || (a != 6) && (a > 12)

TRUE || (a != 6) && (a > 12)

TRUE || FALSE && (a > 12)

TRUE || FALSE && (a > 12)

TRUE && (a > 12)

TRUE && FALSE

TRUE && FALSE

FALSE
```

Przykład
Napisz program znajdujący minimalną z trzech liczb *a, b, c* wykorzystujący zmienną pomocniczą x.

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

using namespace std;

int main()
{
    float a, b, c, x;
    cout << "Podaj pierwsza liczbe: ";
    cin >> a;
    cout << "Podaj druga liczbe: ";
    cin >> b;
    cout << "Podaj trzecia liczbe: ";
    cin >> c;
    x = a;
    if (x > b)
    {
        x = b;
    }
    if (x > c)
    {
        x = c;
    }
    cout << endl;
    cout << "Najmniejsza wprowadzona liczba to: " << x;
    cout << endl;
    system ("pause");
    return 0;
}
```

Uwaga:

Niektóre proste instrukcje `if – else` można zastąpić operatorem warunkowym „`?:`”. Jest to jedyny w językach `C/C++` operator trójargumentowy, którego postać jest następująca:

```
wyrażenie_1 ? wyrażenie_2 : wyrażenie_3
```

Najpierw sprawdzana jest wartość `wyrażenie_1`. Jeśli jest różne od zera (prawdziwe), wówczas wynikiem działania całej instrukcji jest `wyrażenie_2`, w przeciwnym wypadku wynikiem jest `wyrażenie_3`.

Przykład
Zastąpić poniższą instrukcję `if – else` operatorem warunkowym **`?:`**.

```
if (a > b)
 c = a;
else
 c = b;
```

Rozwiązanie:

```
c = (a > b) ? a : b;
```

PĘTLE PROGAMOWE

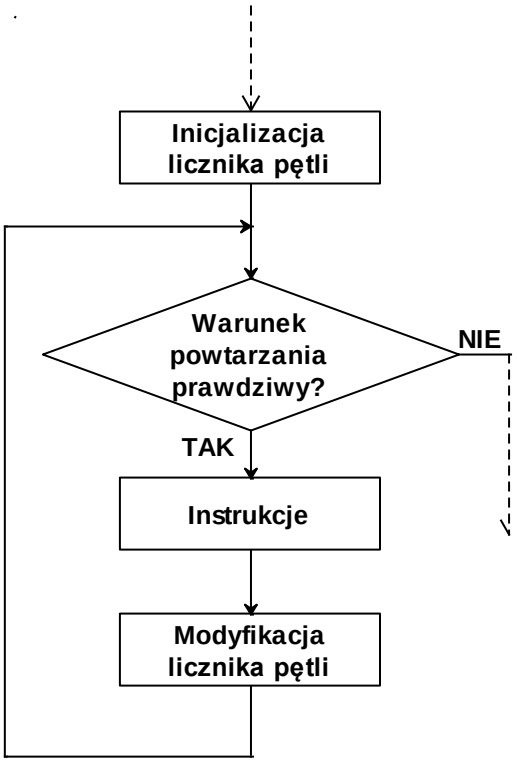
Język C/C++ udostępnia trzy pętle programowe: `for`, `while` oraz `do-while`, zwane inaczej pętlami iteracyjnymi (ang. *iteration* – powtarzanie). Pętlę `for` stosuje się wyłącznie wtedy, gdy liczba powtórzeń (iteracji) jest z góry znana. Natomiast pętle `while` i `do-while` wykonywane są aż do spełnienia określonych warunków. W związku z tym liczba powtórzeń tych pętli nie musi być z góry znana.

Pętla `for`

W pętli `for` nagłówek sterujący pracą pętli (ang. *loop control header*) ma następującą strukturę:

```
for (licznik = wartość_początkowa; warunek_powtarzania; modyfika-
cja_licznika)
 {
  Instrukcje oddzielone średnikami;
 }
```

Można to przedstawić za pomocą następującego schematu blokowego:



Nadanie wartości początkowej licznikowi pętli nosi nazwę inicjalizacji, a warunek powtarzania odpowiada za zakończenie pracy w pętli – jeśli jest fałszywy, następuje wyjście z pętli. W każdym kroku pętli musi następować modyfikacja licznika pętli o wartość zadaną przez programistę. Zmienna, którą wykorzystuje się do sterowania pętlą `for`, musi być typu całkowitego i nazywana jest iteratorem. Jeśli w pętli `for` opuszczona zostanie zmienna sterująca, czyli instrukcja będzie miała postać `for(;;)`, wtedy automatycznie wartość logiczna warunku powtarzania zostanie ustalona na stałe jako `1` (prawda), co oznacza, że pętla będzie wykonywana bezwarunkowo w nieskończoność.

Przykład
Napisz program z pętlą `for`, który wypisuje na ekranie 10 kolejnych liczb całkowitych.

```
#include <iostream>

using namespace std;
```

```
int main()
{
 for (int i=1; i<=10; i++)
   {
    cout << i <<endl;
   }
 return 0;
}
```

Uwagi:

1. Język C/C++ umożliwia również zwiększanie lub zmniejszanie licznika nie o jeden, lecz o inną wartość całkowitą. Wówczas nagłówek pętli może wyglądać następująco:

   ```
   for (i = 4; i < 100; i = i + 2)
   ```

   lub

   ```
   for (i = 1000; i < 100; i = i - 15)
   ```

2. Operacje przypisania `i = i + 2` oraz `i = i - 15` można zapisać w skróconej formie, tj. `i+=2` oraz `i-=15`. Uproszczony zapis może być stosowany w innych, także bardziej skomplikowanych operacjach przypisania. Dla przykładu, poniższe operacje:

   a) $a = a * (4 * b - 1);$

   b) $m = m\%(3 - n/2);$

   c) $b = b - (3 + d * 4);$

   d) $k = k + \big((a = a * 2) - (b = b + 3)\big);$

   można uprościć do postaci:

   a) $a *= 4 * b - 1;$

   b) $m\% = 3 - n/2;$

   c) $b -= 3 + d * 4;$

   d) $k += (a = a * 2) - (b = b + 3);$

Przykład
Napisz program wypisujący liczby od 10 do 0 („końcowe odliczanie”) z jednosekundową przerwą pomiędzy każdą liczbą i komunikatem „KONIEC ODLICZANIA”. Przed rozpoczęciem odliczania i po każdym wyświetleniu liczby ekran powinien być wyczyszczony.

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

using namespace std;

int main()
{

 for (int i=10; i>=0; i--)
  {
```

```
    Sleep(1000);
    system("cls");
    cout << i <<endl;
  }
 cout << endl << "KONIEC ODLICZANIA!"<<endl;

 getch();
 return 0;
}
```

Uwagi:

1. Użyta w programie instrukcja `getch();` ma tutaj podobne działanie do wcześniej używanej instrukcji `getchar();`, czyli wstrzymuje działanie programu do czasu naciśnięcia dowolnego klawisza (nie tylko `ENTER`, jak ma to miejsce w przypadku `getchar();`). Do obsługi `getch();` wymagana jest biblioteka `<conio.h>`, która nie jest w pełni zgodna ze standardem, przez co program staje się mniej przenośny i może nie działać w innych kompilatorach.
2. Zadaniem instrukcji `Sleep(x);`, gdzie `x` jest czasem podanym w milisekundach, jest czasowe wstrzymanie działania programu. Wymaga biblioteki `<windows.h>`. Zauważ, że instrukcja `Sleep` rozpoczyna się od wielkiej litery!
3. Instrukcja system `system("cls");` „czyści” ekran (ang. *clear screen*). Wymaga biblioteki `<windows.h>`.

Przykład
Napisz program, który oblicza silnię liczby n ($n! = 1 \cdot 2 \cdot \cdots n$).

```
#include <iostream>
#include <windows.h>
using namespace std;

int main()
{
    int i, n;
    long long int silnia=1;
    cout << "Program oblicza n!" << endl;
    cout << "Podaj n>=1: ";
    cin >> n;
    for(i=1; i<=n; i++)
    {
        silnia=silnia*i;
    }
    cout << endl << n <<"! = " << silnia << endl << endl;

    system("pause");
    return 0;
}
```

Uwaga:
Program oblicza silnię w sposób iteracyjny. Istnieje też inny sposób, rekurencyjny.

Przykład
Napisz program, którego zadaniem jest wygenerowanie *n-tego* wyrazu ciągu Fibonacciego.

```
#include <iostream>
#include <cstdio>

using namespace std;
```

```
int main()
{
 int i, n;
 unsigned long long int f1, f2, fn;
  cout<<"Ktory wyraz ciagu Fibonacciego obliczyc? ";
  cin>>n;
   f1=0;
   f2=1;
   if (n==1)
    {
      cout<<"Wartosc wyrazu numer "<<n<<" wynosi: "<<f1;
    }
   else if (n==2)
    {
      cout<<"Wartosc wyrazu numer "<<n<<" wynosi: "<<f2;
    }
     else
     {
      for (i=1; i<=n-2; i++)
       {
        fn=f1+f2;
        f1=f2;
        f2=fn;
       }

cout<<"Wartosc wyrazu numer "<<n<<" wynosi: "<<fn;}

getchar();
return 0;
}
```

Przykład
Napisz program losujący sześć liczb z przedziału 1...49.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <conio.h>
#include <iomanip>

using namespace std;

main()
{
 int liczba, n=6, i;
 srand(time(NULL));
  for (i=1; i<=n; i++)
  {
   liczba=1+rand()%49;
   cout<<setw(2)<<liczba<<endl;
  }

 cin.get();
 return 0;
}
```

Uwagi:

1. Funkcja `cin.get();` ma działanie podobne do `getchar();`, czyli wymusza naciśnięcie klawisza `ENTER`.

2. Do formatowania strumieniowych operacji wejścia/wyjścia mogą służyć tzw. manipulatory, które są dostępne po dołączeniu biblioteki <iomanip>. W tabeli zestawiono ich rodzaje i działanie:

Rodzaj manipulatora	Działanie manipulatora
setprecision(x)	Określa precyzję liczb rzeczywistych jako równą x (x – liczba całkowita)
setw(x)	Wyznacza szerokość pola jako równą x znaków (x – liczba całkowita)
setfill(char a)	Ustala, że wolne miejsca zostaną wypełnione znakami a
endl	Umieszcza w strumieniu znak końca wiersza i opróżnia strumień
ends	Umieszcza znak '\0' na końcu łańcucha
dec	Włącza konwersję dziesiętną
hex	Włącza konwersję szesnastkową
oct	Włącza konwersję ósemkową

3. Funkcje przeznaczone do generowania liczb losowych znajdują się w bibliotekach <cstdlib> oraz <ctime>, a opis poleceń i przykłady zastosowań znajdują się w poniższych tabelach:

Polecenia stosowane przy generowaniu liczb losowych

Polecenie	Opis działania
rand()	Funkcja generująca losową liczbę całkowitą zawartą między 0 a RAND_MAX
RAND_MAX	Predefiniowana stała, maksymalna wartość generowana przez funkcję rand(). Zależy od kompilatora i bibliotek
srand(time(NULL))	Funkcja inicjalizująca funkcję rand(), przy każdym uruchomieniu programu daje inną sekwencję liczb losowych
time(NULL)	Odczytany z zegara czas (w sekundach), jaki upłynął od 1970 roku, stanowi wartość bazową przy generowaniu liczb losowych

Przykłady zastosowań funkcji rand() do generowania liczb losowych

Przykład użycia	Wynik działania
liczba=p+rand()%(q-p+1);	Wygenerowanie liczby losowej całkowitej z przedziału [p,q]
liczba=rand()%(q+1)	Wygenerowanie liczby losowej naturalnej z przedziału [0,q]
liczba=p+(double)rand()/RAND_MAX*(q-p)	Wygenerowanie liczby losowej rzeczywistej z przedziału [p,q]
liczba=p+(double)rand()/RAND_MAX	Wygenerowanie liczby losowej rzeczywistej z przedziału [0,1]
liczba=p+(double)rand()/RAND_MAX*q	Wygenerowanie liczby losowej rzeczywistej z przedziału [0,q]

Podobnie jak inne języki programowania, język C/C++ umożliwia stosowanie pętli zagnieżdżonych for.

Przykład
Napisz program wyświetlający tabliczkę mnożenia 10x10.

```
#include <iostream>
#include <cstdio>

using namespace std;

int main()
{

 for (int i = 0; i < 10; i++)
 {
  cout << endl;
   for (int j = 0; j < 10; j++)
    {
```

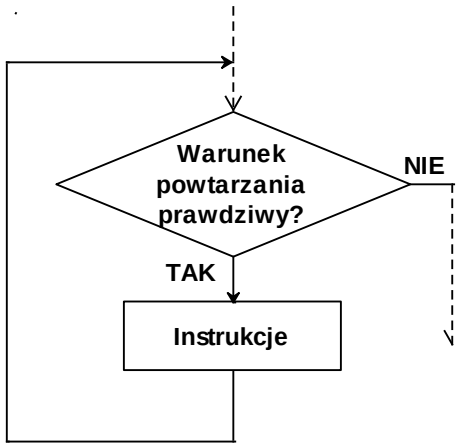
```
     cout << "" << i + 1 << " * " << j + 1 << "" << " = "
          << (i + 1) * (j + 1) << endl;
    };
 };

 getchar();
 return 0;
}
```

Pętla `while`

Pętla `while` ma następującą składnię:

```
while(warunek)
 {
  Instrukcje odzielone średnikami;
 }
```

i można przedstawić ją za pomocą poniższego schematu blokowego.



Ponieważ najpierw sprawdzany jest warunek powtarzania pętli, który decyduje o wykonywaniu pętli, wynika stąd wniosek, że pętla może nie wykonać się ani jednego razu – jeśli warunek od razu jest fałszywy.

Przykład
Używając liczb całkowitych napisz program przeliczający temperaturę wyrażoną w stopniach Celsjusza T_C w zakresie od 0° do 100° na skalę Fahrenheita T_F z krokiem 10° według wzoru:

$$T_F = 32 + \frac{9}{5} \cdot T_C$$

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

using namespace std;

int main()
{
 int celsius, fahr;
 int lower, upper, step;
 lower = 0;     /* dolna granica temperatur */
 upper = 100;   /* górna granica temparatur */
 step = 10;     /* rozmiar kroku */
 celsius = lower;
```

```
  while (celsius <= upper)
   {
    fahr = 32 + 9 * celsius / 5;
    cout << celsius << "\t" << fahr << endl;
    celsius = celsius + step;
   }

getch();
return 0;
}
```

Uwagi:

1. Działanie pętli `while` jest następujące. Najpierw sprawdzany jest warunek zawarty w nawiasach (`celsius <= upper`). Jeśli jest prawdziwy, to w pętli wykonywane są trzy instrukcje zawarte w nawiasach klamrowych. Potem następuje powrót do sprawdzenia warunku itd. aż do momentu, gdy warunek stanie się fałszywy.
2. Kolejność działania na liczbach całkowitych ma duże znaczenie. Gdyby zamiast równania: `32 + 9 * celsius / 5;` zastosować: `32 + 9 / 5 * celsius;`, wówczas ułamek 9/5 najpierw zostałby zaokrąglony do wartości `1` poprzez odrzucenie części ułamkowej (czyli w rezultacie powstałoby działanie: `32 + 1 * celsius`) i program podawałby złe wyniki!
3. Zadaniem instrukcji `cout << celsius << "\t" << fahr << endl;` jest wydruk stopni Celsjusza i Fahrenheita w dwóch kolumnach, oddzielonych od siebie na odległość tabulatora, co zapewnia parametr formatowania napisów `"\t"`. Istnieją też inne sposoby i operatory formatowania napisów (są to tzw. znaki specjalne), które zgromadzono w tabeli:

Znak specjalny	Działanie znaku specjalnego
`\n`	*new line* (przejście do nowej linii)
`\t`	*tab* (tabulator poziomy)
`\v`	*vertical tab* (tabulator pionowy)
`\b`	*backspace* (przesunięcie kursora o jeden znak w lewo)
`\r`	*carriage return* (przesunięcie kursora na początek bieżącego wiersza)
`\'`	apostrof (wypisanie znaku zastrzeżonego – apostrof)
`\"`	cudzysłów (wypisanie znaku zastrzeżonego – cudzysłów)
`\?`	pytajnik (wypisanie znaku zastrzeżonego – pytajnik)
`\\`	*backslash* (wypisanie znaku zastrzeżonego – backslash)

4. Program przeliczający temperatury można także napisać wykorzystując pętlę `for`:

```
for (celsius = lower; celsius <= upper; celsius = celsius + step)
   {
    fahr = 32 + 9 * celsius / 5;
    cout << celsius << "\t" << fahr << endl;
   }
```

Przykład
Napisz przykład przeliczania temperatur z uwzględnieniem liczb rzeczywistych.

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

using namespace std;

int main()
{
 double celsius, fahr;
 int lower, upper, step;
 lower = 0;     /* dolna granica temperatur */
 upper = 100;   /* górna granica temparatur */
 step = 10;     /* rozmiar kroku */
 celsius = lower;

  while (celsius <= upper)
   {
    fahr = 32 + 9 / 5 * celsius;
    cout << fixed << setprecision(2) << setw(6) << celsius
         << setw(10) << fahr << endl;
    celsius = celsius + step;
   }

getch();
return 0;
}
```

Uwagi:

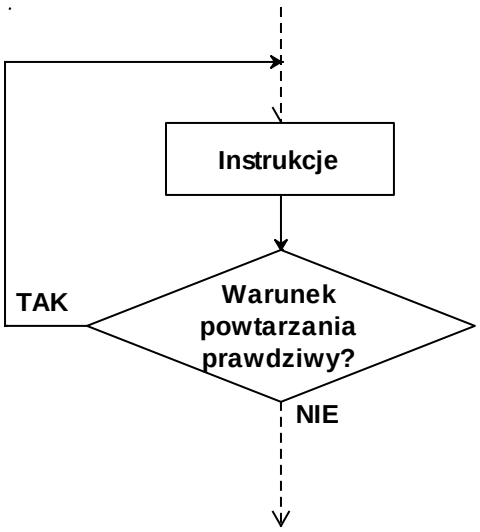
1. Manipulator `fixed` ustala, że liczby zmiennoprzecinkowe (rzeczywiste) będą wyświetlane w standardowy sposób, czyli np. jako `3.482`. Manipulator `fixed` stosowany jest na ogół razem z manipulatorem `setprecision(x)`, którego zadaniem jest wymuszenie wyświetlenia danej liczby z dokładnością `x` miejsc po przecinku.
2. Inny sposób wyświetlania liczb rzeczywistych to notacja naukowa, do której używa się manipulatora `scientific`. W tym wypadku np. liczba `32` zostanie wyświetlona jako `3.200000e+001`, czyli zostanie przedstawiona w notacji wykładniczej jako $3.2 \cdot 10^1$ z dokładnością `6` miejsc po przecinku, chyba że użyjemy manipulatora `setprecision(x)` i ustalimy inną dokładność wyświetlania.

Pętla `do-while`

Składnia pętli `do-while` jest następująca:

```
do
 {
  Instrukcje oddzielone średnikami;
 }
while(warunek);
```

co można przedstawić za pomocą poniższego schematu blokowego.



W wypadku tej pętli najpierw wykonywana jest instrukcja (instrukcje) programu, a dopiero później sprawdzany jest warunek powtarzania pętli. Wynika stąd wniosek, że pętla wykona się przynajmniej jeden raz, nawet jeśli warunek powtarzania pętli od razu jest fałszywy.

Przykład
Zmodyfikuj fragment programu przeliczającego temperatury zastępując pętlę `while` pętlą `do-while`.

```
.......
do
 {
  fahr = 32 + 9 * celsius / 5;
  cout << celsius << "\t" << fahr << endl;
  celsius = celsius + step;
 }
while (celsius <= upper);
.......
```

Przykład
Napisz program, który do skutku żąda od użytkownika podania poprawnego numeru PIN (`1234`).

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

using namespace std;

int main()
{
  int pin;

  do
  {
    cout << "Podaj PIN: ";
    cin >> pin;
  }
  while(pin != 1234);

  cout << endl << "PIN poprawny" << endl << endl;

  system("pause");
  return 0;
}
```

Przykład
Napisz program, który oblicza sumę dwóch liczb i umożliwia użytkownikowi ponowne obliczenia stosując pętlę `do-while`.

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

using namespace std;

int main()
{
    int a, b;
    char c; //ang. character – znak, litera
    do
    {
        system("cls");
        cout << "Podaj a: ";
        cin >> a;
        cout << "Podaj b: ";
        cin >> b;
        cout << "a + b = " << a + b << endl;
        cout << "Czy chcesz liczyc dalej? <T/N>" << endl;
        cin >> c;
    }
    while ((c == 'T') || (c == 't'));
    system("pause");
    return 0;
}
```

Uwaga:

W warunku wyjścia z pętli (`while`) należy przewidzieć sytuację, że użytkownik wpisze z klawiatury żądaną literę jako małą lub wielką. Stąd konieczność zastosowania sumy logicznej `||`.

Przykład
Napisz program, który w pętli nieskończonej zwraca kod ASCII naciśniętego przez użytkownika klawisza (naciśnięcie ESC kończy).

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

using namespace std;

int main()
{
    char znak;
    cout << "Naciskaj klawisze (ESC=wyjscie)" << endl;
    while(1)
    {
      znak = getch();
      if (znak == 27)  //27 to kod klawisza ESC
      break;
      cout << "znak " << znak << " ma kod dziesietny ASCII "
           << (int)znak << endl;
    }
    return 0;
}
```

Uwagi:

1. Instrukcja `while(1)` oznacza, że pętla będzie wykonywała się w nieskończoność, ponieważ wyrażenie w nawiasie `(1)` oznacza prawdę i nigdy nie ulegnie zmianie. Podobny efekt pętli nieskończonej możemy uzyskać pomijając w pętli for wyrażenie logiczne, czyli `for(;;)`. W tym wypadku automatycznie zostanie przyjęta w pętli `for` wartość logiczna `1`, co oznacza wykonywanie w nieskończoność. Do przerwania działania takich pętli służy właśnie instrukcja `break`.
2. Zadaniem instrukcji `znak = getch();` jest przechwycenie znaku wprowadzonego przez użytkownika z klawiatury, ale bez wyświetlania go na ekranie. Jej działanie jest więc nieco inne, aniżeli działanie instrukcji `cin >> znak;`.
3. Zmienna `znak` jest typu `char`, czyli reprezentowana jest przez znak (literę) zapisaną w kodzie `ASCII`. Aby uzyskać jej wartość liczbową, konieczne jest tzw. `rzutowanie`, realizowane przez instrukcję `(int)znak`, która nakazuje wyświetlenie wprowadzonego znaku nie jako np. litery, ale jako liczby, odpowiadającej jej kodowi dziesiętnemu `ASCII`. Zatem instrukcja `cout >> znak;` wyświetli na ekranie wprowadzony przez użytkownika znak, natomiast instrukcja `cout >> (int)znak;` wyświetli dziesiętny kod `ASCII` tego znaku.

Instrukcje `continue, break i switch-case`

Przy przetwarzaniu dużych ilości danych stosuje się zwykle różne pętle programowe. Prócz prostych pętli można tworzyć konstrukcje bardziej złożone. W języku C/C++ przewidziane są do tych celów dodatkowe instrukcje:

Instrukcja `break`

Instrukcja `break` powoduje natychmiastowe bezwarunkowe opuszczenie pętli dowolnego typu i przejście do najbliższej instrukcji po zakończeniu pętli. W przykładzie poniżej nieskończoną pętlę przerywa instrukcja `break` po podaniu z klawiatury wartości `0`.

Przykład
Napisz program sumujący kolejno wprowadzane przez użytkownika liczby rzeczywiste. Wprowadzenie przez użytkownika liczby `0` kończy obliczenia.

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

using namespace std;

int main()
{
    double a, suma = 0;
    while(1)
     {
      cout << "Podaj liczbe do sumowania: ";
      cin >> a;
      if (a == 0)
       {
        cout << endl << "Przerwano sumowanie";
        break;
       }
      suma += a;
     }

     cout << endl << "Suma wprowadzonych liczb wynosi: " << suma;

    getch();
    return 0;
```

```
}
```

Uwaga:

Język C/C++ oferuje także inną instrukcję, która służy do przerywania działania dowolnego fragmentu programu. Jest nią instrukcja `goto`, której składnia jest następująca: `goto etykieta;`. Jej używanie w zdecydowanej większości przypadków świadczy o złych nawykach programistycznych. Istnieją jednak bardzo rzadkie sytuacje, w których instrukcja `goto` może mieć zastosowanie, tj. w pętlach wielokrotnie zagnieżdżonych. W przeciwieństwie bowiem do instrukcji `break`, która przerywa działanie tylko bieżącej pętli, za pomocą instrukcji `goto` można opuścić od razu wszystkie pętle zagnieżdżone, przykładowo:

```
int i, j, k;
while(i>0)
 {
  while(j<10)
   {
    for(k=1; k<10; k++)
     {
      //.............. kolejne instrukcje lub petle
      if(blad) goto koniec;
     }
   }
koniec:
cout << "Wystapil blad" << endl;
```

Jeśli w trakcie wykonywania kolejnych instrukcji zmienna `blad` przybierze wartość niezerową, wtedy nastąpi natychmiastowe przerwanie wykonywania tej części programu oraz przejście do miejsca w programie oznaczonego etykietą `koniec:`. Kolejne instrukcje programu będą wykonywane począwszy od tej etykiety.

Przykład
Zmodyfikuj program weryfikujący PIN w ten sposób, aby po 3 nieudanych próbach został wygenerowany komunikat „Karta zablokowana”.

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

using namespace std;

int main()
{
  int i = 1, pin;

  do
  {
    cout << "Podaj PIN: ";
    cin >> pin;
    if (pin == 1234) break;
    i++;
  }
  while(i <= 3);

  if (pin == 1234)
  {
```

```
      cout << endl << "PIN poprawny" << endl << endl;
  }

  else
  {
      cout << endl << "Karta zablokowana!" << endl << endl;
  }


  system("pause");
  return 0;
}
```

Instrukcja `continue`

Instrukcja `continue` powoduje przedwczesne, bezwarunkowe zakończenie wykonania wewnętrznej instrukcji pętli i podjęcie próby realizacji następnego cyklu pętli. Jest to jedynie próba, ponieważ najpierw zostanie sprawdzony warunek kontynuacji pętli.

Przykład
Zmodyfikuj program sumujący liczby w taki sposób, aby:

- jeśli liczba jest dodatnia – dodawał ją do sumy;
- jeśli liczba jest ujemna – nie robił nic, tylko pomijał bieżącą pętlę przy pomocy instrukcji `continue`;

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

using namespace std;

int main()
{
    double a, suma = 0;
    cout << "Sumuje tylko liczby dodatnie" << endl << endl;
    for(;;)
     {
      cout << "Podaj liczbe do sumowania: ";
      cin >> a;
      if (a < 0) continue;
      if (a == 0)
      {
        cout << endl << "Przerwano sumowanie";
        break;
      }
      suma += a;
      }
    cout << endl << "Suma wprowadzonych liczb wynosi: " << suma;

    getch();
    return 0;
}
```

Instrukcja `switch-case`

Instrukcja `switch` („przełącz”) dokonuje wyboru w zależności od stanu wyrażenia przełączającego (*selector*) jednego z możliwych przypadków – wariantów (*case*). Każdy wariant jest oznaczony za pomocą stałej – tzw. etykiety wyboru. Wyrażenie przełączające w C++ może przyjmować wartości typu `int`. Ostatnia instrukcja, o nazwie `default`, ma za zadanie wyświetlić odpowiedni komunikat w sytuacji, gdy użytkownik wybierze selektor spoza przewidzianych przez programistę przypadków. Ogólna postać instrukcji `switch-case` jest następująca:

```
switch (selector)
{
case STAŁA1: Ciąg_instrukcji-wariant 1; break;
case STAŁA2: Ciqg_instrukcji-wariant 2; break;
case STAŁAn: Ciąg_nstrukcji-wariant n; break;
default    : Ostatni_ciąg_instrukcji;
}
```

Uwaga:
Jeśli w wariantach zostałyby pominięte instrukcje `break`, to po dokonaniu wyboru i skoku do odpowiedniej etykiety, wykonane zostałyby również wszystkie instrukcje poniżej tej etykiety, aż do klamry zamykającej blok `switch-case`.

Przykład
Napisz program, którego zadaniem jest wyświetlenie nazwy dnia tygodnia na podstawie jego numeru (poniedziałek jest tutaj pierwszym dniem tygodnia). Sprawdź działanie programu po usunięciu instrukcji `break`.

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

using namespace std;

int main()
{
    int numer_dnia;
    cout << "Podaj numer dnia tygodnia: ";
    cin >> numer_dnia;
    cout << endl;
    switch (numer_dnia)
    {
     case 1: cout << "PONIEDZIALEK"; break;
     case 2: cout << "WTOREK"; break;
     case 3: cout << "SRODA"; break;
     case 4: cout << "CZWARTEK"; break;
     case 5: cout << "PIATEK"; break;
     case 6: cout << "SOBOTA"; break;
     case 7: cout << "NIEDZIELA"; break;
     default: cout << "NIE MA TAKIEGO DNIA TYGODNIA!";
    }

 cout << endl;
 getch();
 return 0;
}
```

Zadanie
Napisz identyczny do powyższego program, ale stosując zamiast `switch-case` zagnieżdżone instrukcje `if-else`.

Przykład
Napisz program działający jako prosty kalkulator czterodziałaniowy.

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

using namespace std;
```

```
int main()

{
    double liczba1, liczba2, wynik;
    char literka, znak;
    do
    {
        cout << "Podaj liczbe, nastepnie znak [+,-,*,/] i druga liczbe."
             << endl;
        cout << "Na koniec nacisnij ENTER." << endl;
        cin >> liczba1 >> znak >> liczba2;

    switch(znak)
    {
      case '+' : wynik = liczba1 + liczba2;
                 break;
      case '-' : wynik = liczba1 - liczba2;
                 break;
      case '*' : wynik = liczba1 * liczba2;
                 break;
      case '/' : if (liczba2 == 0)
                 {
                    cout << "Blad: Dzielenie przez zero!";
                    getch();
                    exit(1);
                 }
                 wynik = liczba1 / liczba2;
                 break;
      default: cout << "Blad: Niewlasciwy znak dzialania!";
               getch();
               exit(1);
               break;
    }

     cout << liczba1 << znak << liczba2 << "=" << wynik;
     cout << endl << "Liczymy jeszcze raz? (T/N):";
     literka=getch();
     system("cls");
  } while ((literka == 't') || (literka == 'T'));

    return 0;
}
```

Uwaga:
Funkcja `exit` powoduje zakończenie działania (0 – normalne, 1 – w wyniku błędu).

TABLICE

Tablice jednowymiarowe

Tablica – ciąg ułożonych kolejno w pamięci elementów tego samego typu.

Jeśli zachodzi potrzeba użycia i zapamiętania większej liczby zmiennych tego samego typu, najlepszym rozwiązaniem okazuje się najczęściej tablica (ang. *array*), która jest złożoną strukturą danych i stanowi kontener (pojemnik) na takie dane. Tablica składa się z komórek i zajmuje w pamięci operacyjnej ciągły obszar pamięci, tzn. komórki umieszczone są w pamięci jedna obok drugiej.

Każda komórka tablicy posiada swój unikalny numer, który nazywany jest indeksem tablicy. Rozmiar tablicy może być z góry ustalony (tablice statyczne) lub może zmieniać się w trakcie wykonywania programu (tablice dynamiczne). Tablice mogą być jedno-, dwu- lub wielowymiarowe.

Podobnie jak inne zmienne w języku C++, tablica przed pierwszym użyciem musi być właściwie zadeklarowana, przykładowo:

```
int liczby[5]; //tablica o nazwie liczby, składająca się z pięciu zmiennych typu całkowitego
char slowo[10]; //tablica o nazwie slowo, składająca się z dziesięciu zmiennych typu znakowego
```

Wygląd jednowymiarowej tablicy `liczby`, niewypełnionej jeszcze danymi, będzie więc następujący:

liczby[0]	liczby[1]	liczby[2]	liczby[3]	liczby[4]

Jak widać, w języku C++ numeracja komórek tablicy zaczyna się od indeksu 0, zatem w tablicy składającej się z 5 komórek pierwsza będzie miała numer 0, a ostatnia numer 4. Częsty błąd polega na zapominaniu o tej właściwości. Jeśli odwołamy się do komórki o indeksie 5 (o ile system operacyjny na to pozwoli), możemy zniszczyć zawartość komórki pamięci, która nie należy do naszej tablicy i w rezultacie spowodować trudne do zidentyfikowania błędy.

Tablicę możemy wypełnić danymi (zainicjalizować) na wiele sposobów już w momencie deklaracji, na przykład:

```
int liczby[5]={4, 1, 2, 0, 3};
```

liczby[0]	liczby[1]	liczby[2]	liczby[3]	liczby[4]
4	1	2	0	3

```
int liczby[5]={4, 1};
```

liczby[0]	liczby[1]	liczby[2]	liczby[3]	liczby[4]
4	1	0	0	0

```
int liczby[]={4, 1, 2, 0};
```

liczby[0]	liczby[1]	liczby[2]	liczby[3]
4	1	2	0

```
int liczby[5]={0};
```

liczby[0]	liczby[1]	liczby[2]	liczby[3]	liczby[4]
0	0	0	0	0

W przedostatnim przypadku rozmiar tablicy został ustalony na cztery komórki na podstawie danych inicjalizacyjnych. Dane do komórki tablicy `liczby` możemy wpisywać za pomocą instrukcji przypisania, np.:

```
liczby[0]=12;
```

Dane w komórkach tablicy `liczby` możemy modyfikować na różne sposoby, przykładowo:

`liczby[0]++;` *//zwiększenie zawartości komórki* `liczby[0]` *o jeden*

`liczby[4]-=2;` *//zmniejszenie zawartości komórki* `liczby[4]` *o dwa*

`liczby[1]%=4;` *//zastąpienie zawartości komórki* `liczby[1]` *resztą z podzielenia jej przez cztery*

`liczby[3]*=liczby[2]+7;` *//zawartość komórki* `liczby[3]` *zostanie przemnożona przez sumę komórki* `liczby[2]` *oraz liczby* `7`

Wypełnianie komórek tablicy liczby stałą wartością jest bardzo proste.

Przykład
Napisz program wypełniający 5-elementową tablicę `liczby` stałą wartością 7, a następnie wypisz jej zawartość na standardowym urządzeniu wyjściowym.

```
#include <iostream>
#include <cstdio>

using namespace std;

const int N = 5;

int main()
{
 int liczby[N], i;
 for(i = 0; i < N; i++)
  liczby[i] = 7;
 for(i = 0; i < N; i++)
  cout << "liczby[" << i << "] = " << liczby[i] << endl;

    getchar();
    return 0;
}
```

Uwaga:
Deklaracja stałej `const int N = 5;` (ang. *constant* – stała) oznacza, że każde wystąpienie stałej `N` w programie zostanie zastąpione liczbą `5`. Jest to bardzo wygodne w przypadku, gdy `N` występuje w programie wielokrotnie. W razie potrzeby zmiany wartości `N`, wystarczy uczynić to jeden raz – w deklaracji stałej. Nie wolno przy tym manipulować wartością stałej `N` w dalszej części programu – takie działanie kompilator uzna za błąd, ponieważ wartość `N` przewidziana jest tylko do odczytu. W języku C++ przyjęło się również, że oznaczenie stałej piszemy samymi wielkimi literami (tutaj: `N`).

Tablicę `liczby` możemy wypełnić także ręcznie, korzystając z pętli `for`.

Przykład
Napisz program, który pozwala użytkownikowi wprowadzić liczby do 5-elementowej tablicy `liczby`, a następnie wypisuje zawartość tablicy na standardowym wyjściu.

```
#include <iostream>
#include <cstdio>

using namespace std;

const int N = 5;
```

```
int main()
{
 int liczby[N], i;
 for(i = 0; i < N; i++)
  {
      cout << "Wartosc liczby[" << i << "] = ";
      cin >> liczby[i];
  }
 cout << endl;

 for(i = 0; i < N; i++)
      cout << "Wprowadziles liczby[" <<i <<"] = " << liczby[i] << endl;

    getchar();
    return 0;
}
```

Przykład
Napisz program wykorzystujący tablicę i obliczający $n - ty$ wyraz ciągu Fibonacciego. Sprawdź, jaką największą liczbę całkowitą jest w stanie wyświetlić komputer – bez stosowania zapisu naukowego, czyli zapisu w rodzaju $1{,}602392 \cdot 10^{141}$.

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

using namespace std;

int main()
{
    int n;
    long double fib[100000];
    cout << setprecision(100000);

    cout << "Ktory wyraz ciagu Fibonacciego mam wyznaczyc? ";
    cin >> n;

    fib[0] = 1;
    fib[1] = 1;

    for (int i = 2; i < n; i++)
    {
        fib[i] = fib[i-1] + fib[i-2];
    }

    cout << endl << "Wyraz ciagu Fibonacciego nr "<< n <<" wynosi:" << endl
<< endl << fib[n-1] << endl;

    getchar();
    return 0;
}
```

Uwaga:
Rozwiązaniem zadania jest wyraz ciągu Fibonacciego o numerze `23601`, za którym stoi olbrzymia liczba całkowita. Dowiadujemy się o tym dzięki zastosowaniu typu `long double` oraz manipulatora `setprecision(100000)`, którego obsługą zajmuje się biblioteka `iomanip`. Liczba w nawiasie mówi nam, ile maksymalnie cyfr liczby całkowitej ma się znaleźć na ekranie. Jeśli zbytnio ograniczymy liczbę cyfr, wynik zostanie skrócony i wyświetlony w formacie naukowym. Przykładowo, gdy ograniczymy liczbę cyfr do `10`, to dla `23601.` wyrazu ciągu otrzymamy na ekranie liczbę `9.285654867e+4931`, czyli $9{,}285654867 \cdot 10^{4931}$.

Oprócz wypełniania tablic danymi, często występuje potrzeba wstawienia w określone miejsce tablicy jakiegoś elementu. Oznacza to konieczność przesunięcia pozostałych elementów o jeden w kierunku rosnących indeksów, do czego wykorzystujemy kopiowanie. Jeśli ostatnia komórka tablicy była wcześniej zajęta, jej element zostanie w ten sposób zniszczony.

Przykład
Chcemy wstawić do tablicy `liczby` wartość `7` do komórki o indeksie `2`.

liczby[0]	liczby[1]	liczby[2]	liczby[3]	liczby[4]
4	1	2	5	3

Zatem najpierw do komórki `liczby[4]` należy skopiować zawartość komórki `liczby[3]`, a do komórki `liczby[3]` należy skopiować zawartość komórki `liczby[2]`. W ten sposób uzyskamy następujący efekt:

liczby[0]	liczby[1]	liczby[2]	liczby[3]	liczby[4]
4	1	2	2	5

Jak widać, chwilowo będziemy mieli zdublowaną wartość komórki `liczby[2]` w komórce `liczby[3]`, ale wstawiając wartość `7` do komórki `liczby[2]` zniszczymy jej pierwotną zawartość. Po zakończeniu operacji wstawiania tablica `liczby` będzie więc wyglądać następująco:

liczby[0]	liczby[1]	liczby[2]	liczby[3]	liczby[4]
4	1	7	2	5

Program realizujący operację wstawiania może wyglądać następująco:

```
#include <iostream>
#include <cstdio>

using namespace std;

int main()
{
 int liczby[5], i;

 //wypełnianie tablicy liczbami
 for(i = 0; i < 5; i++)
  {
      cout << "Wartosc liczby[" << i << "] = ";
      cin >> liczby[i];
  }

  cout << endl;

 //wyświetlenie wprowadzonej tablicy
 for(i = 0; i < 5; i++)
      cout << "Wprowadziles liczby[" <<i <<"] = " << liczby[i] << endl;

 //przesuniecie (skopiowanie) elementów tablicy
 for (i=3; i>=2; i--) liczby[i + 1]=liczby[i];

  liczby[2] = 7; //wstawienie nowego elementu

 cout << endl;

 // wyświetlenie tablicy po modyfikacji
 for(i = 0; i < 5; i++)
```

```
      cout << "Po modyfikacji liczby[" << i <<"] = " << liczby[i] << endl;

    getchar();
    return 0;
}
```

Podobnie działa usuwanie jakiegoś elementu tablicy. W tym wypadku elementy przesuwają się w przeciwnym kierunku, zaś do ostatniej komórki wstawiamy zero, bo inaczej ostatni i przedostatni element miałyby zdublowaną zawartość. Usuńmy, dla przykładu, zawartość komórki `liczby[2]`. Po zakończeniu tej operacji tablica `liczby` powinna mieć wygląd następujący:

liczby[0]	liczby[1]	liczby[2]	liczby[3]	liczby[4]
4	1	5	3	0

Poniżej program realizujący operację usuwania elementu tablicy:

```
#include <iostream>
#include <cstdio>

using namespace std;

int main()
{
 int liczby[5], i;

 //wypełnianie tablicy liczbami
 for(i = 0; i < 5; i++)
  {
      cout << "Wartosc liczby[" << i << "] = ";
      cin >> liczby[i];
  }

  cout << endl;

 //wyświetlenie wprowadzonej tablicy
 for(i = 0; i < 5; i++)
      cout << "Wprowadziles liczby[" <<i <<"] = " << liczby[i] << endl;

 //przesunięcie (skopiowanie) elementów tablicy
 for (i = 2; i <= 3; i++) liczby[i] = liczby[i + 1];

  liczby[4]=0; //wstawienie zera do ostatniej komórki tablicy

 cout << endl;

 // wyświetlenie tablicy po modyfikacji
 for(i = 0; i < 5; i++)
      cout << "Po modyfikacji liczby[" << i <<"] = " << liczby[i] << endl;

    getchar();
    return 0;
}
```

Tablice dwu- i wielowymiarowe

Ogólna deklaracja tablicy dwuwymiarowej jest następująca:

```
typ identyfikator[liczba_wierszy][liczba_kolumn];
```

przykładowo:

```
double wyniki[4][5]; //tablica o nazwie wyniki, składająca się z czterech wierszy i pięciu kolumn
```

Tablica `wyniki` przed wypełnieniem danymi będzie miała wygląd:

wyni- ki[0][0]	wyni- ki[0][1]	wyni- ki[0][2]	wyni- ki[0][3]	wyni- ki[0][4]
wyni- ki[1][0]	wyni- ki[1][1]	wyni- ki[1][2]	wyni- ki[1][3]	wyni- ki[1][4]
wyni- ki[2][0]	wyni- ki[2][1]	wyni- ki[2][2]	wyni- ki[2][3]	wyni- ki[2][4]
wyni- ki[3][0]	wyni- ki[3][1]	wyni- ki[3][2]	wyni- ki[3][3]	wyni- ki[3][4]

Dostęp do poszczególnych komórek tej tablicy jest identyczny, jak w tablicy jednowymiarowej z tą różnicą, że trzeba użyć teraz dwóch indeksów. Jeśli chcemy wstawić na przykład wartość `3.75` do komórki leżącej na przecięciu drugiego wiersza i trzeciej kolumny, należy po prostu zastosować instrukcję przypisania:

```
wyniki[1][2]=3.75;
```

W języku C++ możliwe jest także użycie tablic wielowymiarowych. Przykładowa deklaracja tablicy trójwymiarowej jest następująca:

```
int kostka[3][3][3];
```

Dochodzi tutaj trzeci wymiar (podobnie jak w układzie współrzędnych `x`, `y`, `z`). Zatem tablica `kostka` składa się z trzech tablic dwuwymiarowych, każda o rozmiarze `3x3`, umieszczonych jakby jedna za drugą. Dostęp do komórek tej tablicy jest taki sam, jak w innych tablicach, czyli za pomocą indeksów (tutaj trzech). Na tej samej zasadzie tworzy się tablice posiadające więcej niż trzy wymiary. W praktyce programistycznej najczęściej zachodzi jednak potrzeba używania tablic jedno- i dwuwymiarowych.

Wskaźniki i referencje w języku C++

Wskaźnik jest to zmienna, która zawiera adres innej zmiennej umieszczonej w pamięci RAM. Adres w pamięci RAM jest liczbą i w związku z tym można ją wykorzystywać do różnych celów. Adresami rządzą specyficzne prawa, dlatego w języku C++ występuje specjalny typ zmiennych – tzw. zmienne wskazujące (ang.*pointers*). Pojęcia „komórka pamięci" i „bajt" są rozróżniane, ponieważ obszar zajmowany w pamięci przez zmienną może mieć różną długość. Każdy wskaźnik musi więc należeć do jakiegoś typu, przy czym typ wskaźnika musi odpowiadać typowi wskazywanego obiektu.

Aby komputer wiedział, ile kolejnych bajtów pamięci zajmuje wskazany obiekt (liczba długa, krótka, znak itp.), deklarując wskaźnik trzeba podać na co będzie wskazywał. Istnienie wskaźników umożliwia pośrednie odwoływanie się do wskazywanego obiektu (liczby, znaku, łańcucha znaków itp.) a także stosunkowo proste odwołanie się do obiektów sąsiadujących z nim w pamięci.

Główne obszary zastosowania wskaźników to:

1. Efektywniejsza praca z tablicami.
2. Zmiana wartości argumentów przysyłanych do funkcji
3. Dostęp do specjalnych komórek pamięci
4. Rezerwacja obszarów pamięci.

Przykład 1.
Przeanalizuj poniższy kod i uwagi pod nim zawarte.

```
#include <iostream>

using namespace std;

int main(void)
{
  int liczba = 10;
  int *pointer = &liczba;     // (1)
  cout << liczba << endl;     // (2)
  cout << *pointer << endl;   // (3)
  cout << &liczba << endl;    // (4)
  cout << pointer << endl;    // (5)
  return 0;
}
```

Uwagi:

1. Deklaracja zmiennej wskaźnikowej (wskaźnika) o nazwie `pointer` i ustawienie go na zmiennej o nazwie `liczba`
2. Zostanie wyświetlona liczba 10
3. Zostanie wyświetlona liczba 10
4. Zostanie wyświetlony adres (np. `0x28ff08`), pod którym znajduje się zmienna o nazwie `liczba`
5. Zostanie wyświetlony adres, na który wskazuje wskaźnik o nazwie `pointer` (czyli `0x28ff08`)

Przykład 2.
Przeanalizuj poniższy program stosujący wskaźniki w języku C++. Uważnie zapoznaj się z komentarzami pod listingiem.

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

using namespace std;

int main(int argc, char *argv[])
{
int zm1 = 6, zm2 = 3;              // (1)
int *wskaznik;                     // (2)

 wskaznik = &zm1;                  // (3)

 cout << "zmienna = " << zm1       // (4)
      << "\n a odczytana przez wskaznik = "
      << *wskaznik << endl;

 zm1 = 8;                          // (5)
 cout << "zmienna = " << zm1
      << "\n a odczytana przez wskaznik = "
      << *wskaznik << endl;

 *wskaznik = 10;                   // (6)
 cout << "zmienna = " << zm1
      << "\n a odczytana przez wskaznik = "
      << *wskaznik << endl;

 wskaznik = &zm2;                  // (7)
 cout << "zmienna = " << zm1
      << "\n a odczytana przez wskaznik = "
```

```
      << *wskaznik << endl;

 cout << endl;

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

Komentarze:

1. Deklaracja dwóch obiektów `zm1` i `zm2` typu `int` wraz z przypisaniem im wartości początkowych
2. Definicja wskaźnika typu `int` (symbol `*` oznacza tzw. operator wyłuskania – *dereference operator*).
3. Dzięki symbolowi `&` (tzw. operator adresu – *address-off operator*) wskaźnik uzyskuje adres zmiennej o nazwie `zm1`. Możliwy jest także zapis skrócony:

```
int *wskaznik = &zmienna;
```

4. Wypisanie na ekranie wartość tego, na co pokazuje wskaźnik.
5. Przypisanie zmiennej `zm1` wartości `8`, wskutek czego następuje zmiana wartości tej zmiennej oraz konsekwentnie wartości wskazywanej przez wskaźnik, ponieważ jest to ta sama zmienna.
6. Zapis `*wskaznik = 10;` oznacza wpisanie do tego, na co pokazuje wskaźnik (czyli do zmiennej `zm1`) wartości `10`. Na ekranie pojawi się dwukrotnie liczba `10`, ponieważ wskaźnik zmienił zawartość wskazywanego obiektu (czyli `zm1`) na `10`, a następnie wskazał na ten obiekt, którego wartość uległa najpierw zmianie na `10`.
7. Przesunięcie tego samego wskaźnika na obiekt `zm2` i wydrukowanie zawartości tej zmiennej. Ten sam wskaźnik może więc wskazywać inną zmienną – pod warunkiem, że jest tego samego typu.

Przykład 3.
Bez użycia kompilatora ustal, jaką wartość przyjmują zmienne `n`, `rn`, `rr` oraz `*wsk` w trakcie wykonywania poniższego programu oraz po jego zakończeniu.

```
#include <iostream>

using namespace std;

int main()
{
    int n = 10;
    int& rn = n;
    rn = 2;
    int *wsk = &rn;
    int& rr = rn;
    getchar();
    return 0;
}
```

Wyrażenie `&x` jest wskaźnikiem zmiennej `x`. W związku z tym instrukcja

```
wsk_x = &x;
```

nadaje wskaźnikowi `wsk_x` wskazanie zmiennej `x`, a wyrażenie `*wsk_x` jest zmienną `x`. Dla przykładu, instrukcja

```
*wsk_x = 21;
```

nadaje zmiennej `x` wartość `21`, a instrukcje

```
++*wsk_x;
 (*px)++;
```

zwiększają zmienną `x` o `1`, natomiast instrukcja

```
*px++;
```

zwiększa wskaźnik `wsk_x` o `1`, gdyż priorytet operacji ++ jest wyższy niż operator wyłuskania `*`.

Nie wolno przy tym zapomnieć, że zwiększanie (zmniejszanie) wskaźnika np. o jeden nie zwiększa (zmniejsza) adresu pamięci o jeden, ale o rozmiar wskaźnika, który wynika z typu, na jaki wskaźnik wskazuje. Jeśli jest to wskaźnik do typu `int`, to następuje zwiększenie (zmniejszenie) o 4 bajty, jeśli do typu `double`, to o 8 bajtów itd.

Przykład 4.

```
int a, b, c, *ptr = &c, *A[5] = {&a, &b, &c};
```

Zdefiniowany wskaźnik `ptr` wskazuje na zmienną `c`. Na przykład instrukcja `*ptr = 3;` daje ten sam efekt, jak instukcja przypisania `c = 3`. Tablica `A` zawiera `5` wskaźników, z których trzy pierwsze wskazują kolejno na zmienne `a, b, c`, a kolejne dwa są zerowe (czyli równe `NULL`). Tym samym instrukcja `*A[1] = 7;` podstawia wartość `7` pod zmienną `b`.

Przykład 5.

```
char *wsk = "Tekst wskazywany przez wsk";
```

Wskaźnik `wsk` wskazuje na pierwszy znak tekstu. Zatem instrukcja

```
cout << wsk << endl;
```

spowoduje wydruk całego wskazywanego tekstu, a instrukcja

```
cout << *wsk << endl;
```

wydrukuje pierwszą literę tekstu, czyli `T`. Ten sam efekt spowoduje instrukcja

```
cout << wsk[0] << endl;
```

Do wskaźnika można dodawać liczby całkowite, zatem np. instrukcja `wsk + 2` wskazuje na literę `k`, czyli wyrażenie `*(wsk + 2)` jest tożsame z wyrażeniem `wsk[2]` (a w zasadzie kodowi ASCII litery `k`).

Przykład 6.

```
char *PORY_ROKU[] = {"wiosna", "lato", "jesien", "zima"};
```

Tablica `PORY_ROKU` jest tablicą czterech wskaźników do nazw pór roku. Dla przykładu, `PORY_ROKU[1]` wskazuje na `lato`. Ten sam efekt powoduje wyrażenie *(`PORY_ROKU + 1)`. Natomiast wynikiem wyrażenia `PORY_ROKU[1][2]` jest litera `t`.

Przykład 7.

```
double (*A[])(double) = {sin, cos, tan, sqrt, log, log10};
```

Lista tablicy `A` zawiera nazwy funkcji matematycznych. Nazwa funkcji jest stałą wskaźnikową do funkcji. Tak więc tablica `A` zawiera 6 wskaźników do funkcji matematycznych o argumencie i wyniku typu double. Dla przykładu, obliczenie pierwiastka kwadratowego (`sqrt`) można wykonać instrukcją

```
y = (*A[3])(x);
```

Uwaga:
Za pomocą wskaźników można przekazać do funkcji zmienną, a nie tylko jej wartość, ponieważ poprzez przekazane wskazanie funkcja ma dostęp do wskazywanej zmiennej, która zdefiniowana jest na zewnątrz funkcji.

Przykłady użycia wskaźników:

```
float *A;               // wskaźnik zmiennej typu float (*A jest typu
                        float)
int **A;                // wskaźnik wskaźnika zmiennej typu int (**A jest
                        typu int)
char *A[5];             // tablica 5 wskaźników zmiennych typu char
long (*A)[];            // wskaźnik tablicy zmiennych typu long
int *(*A)[];            // wskaźnik tablicy wskaźników zmiennych typu int
double *A(void);        // funkcja zwracająca wskaźnik zmiennej typu double
double (*A)(void);      // wskaźnik funkcji o wartości double
float *(*A)(void);      // wskaźnik funkcji zwracającej wskaźnik typu float
int (*(*A)(void))[];    // wskaźnik funkcji zwracającej wskaźnik tablicy
                        typu int
```

W deklaracjach można inicjować wskaźniki stosując takie same zasady, jak przy inicjowaniu zmiennych.

W języku C++ każdy program otrzymuje do dyspozycji pewien obszar w pamięci operacyjnej zwany kopcem (ang. *heap*), który przeznaczony jest na umieszczenie w nim (alokację) obiektów dynamicznych, czyli obiektów tworzonych i niszczonych w trakcie wykonywania programu. Do tworzenie i niszczenia używa się operatorów `new` oraz `delete`. Pierwszy z nich przydziela (alokuje) pamięć na kopcu, drugi zaś ją zwalnia. Ogólna składnia operatora `new` jest następująca:

```
int *ptr = new typ_danych (wartość początkowa);
```

Natomiast składnia operatora delete ma ogólną postać:

```
delete ptr;
```

Należy zawsze pamiętać o zwalnianiu pamięci zaalokowanej operatorem `new` poprzez użycie operatora `delete`. W przeciwnym wypadku zachodzi niebezpieczeństwo nieokreślonego zachowania programu podczas wykonania. Nie należy również używać zmiennej wskaźnikowej po jej zniszczeniu, gdyż jej wartość staje się wtedy nieokreślona, przez co może spowodować bardzo trudne do wykrycia błędy.

Dynamiczną alokację pamięci można użyć do deklarowania tablic, których rozmiary mogą zmieniać się w trakcie wykonywania programu lub o wielkości których chce decydować użytkownik.

Przykład
Zadeklarować tablicę tworzoną dynamicznie przez użytkownika.

```
#include <iostream>
using namespace std;
int main()
{
  int wielkosc;
  cout << "Podaj wielkosc tablicy" << endl;
  cin >> wielkosc;
  int *tab = new int [wielkosc]; // tworzy tablicę o nazwie tab
  delete [] tab; // nawiasy [] wskazują, że do zniszczenia jest tablica
  return 0;
}
```

Referencja jest z kolei specjalnym, niejawnym wskaźnikiem, który działa podobnie jak alternatywna nazwa dla zmiennej (tzw. alias). Zmiana wartości zmiennej wpływa na wartość skojarzonej z nią zmiennej referencyjnej i odwrotnie. Obie zmienne muszą być tego samego typu. Referencje używane są głównie w charakterze argumentów i wartości zwracanych przez funkcje, chociaż w programie można deklarować i używać referencji niezależnych.

Przykład 8.
Zapoznaj się z programem ilustrującym działanie referencji.

```
#include <iostream>

using namespace std;

int main(void)
{
  int liczba = 10;
  int &ref = liczba;          // deklaracja referencji o nazwie ref
  cout << liczba << endl;     // wyswietli wartosc 10
  cout << ref << endl;        // wyswietli wartosc 10
  cout  <<  &liczba  <<  endl;     //  wyswietli  adres  zmiennej  liczba  np.
0x28ff08
  cout << &ref << endl;       // wyswietli adres zmiennej ref np. 0x28ff08
  return 0;
}
```

Przykład 9.
Zapoznaj się z programem ilustrującym działanie referencji oraz z uwagami pod listingiem.

```
#include <iostream>

using namespace std;

int main()
{
    int zmienna = 0;
    int& ref = zmienna; /* 1 */
    cout << "Wartosc poczatkowa zmiennej i zmiennej referencyjnej" << endl;
    cout << "zmienna = " << zmienna << endl;
    cout << "zmienna referencyjna = " << ref << endl << endl;
    zmienna = 1;
    cout << "Zmiana tylko wartosci zmiennej na 1" << endl;
    cout << "zmienna = " << zmienna << endl;
    cout << "zmienna referencyjna = " << ref << endl << endl;
    ref = 2;
    cout << "Zmiana tylko wartosci zmiennej referencyjnej na 2" << endl;
    cout << "zmienna = " << zmienna << endl;
    cout << "zmienna referencyjna = " << ref << endl;
    getchar();
    return 0;
}
```

Uwagi:

1. Zapis `int& ref = zmienna` oznacza zdefiniowanie zmiennej referencyjnej o nazwie `ref` i przypisanie jej wartości zawartej pod adresem zmiennej o nazwie `zmienna`.
2. Jeśli w linii z komentarzem `/* 1 */` zamienić istniejący wiersz np. na `int ref = 0;`, to z powodu likwidacji skojarzenia zmiennej o nazwie `zmienna` ze zmienną o nazwie `ref` obie zmienne staną się od siebie niezależne.
3.

Tworzenie własnych funkcji w języku C++

Język C++ umożliwia tworzenie własnych funkcji (ang. *function*). Funkcja jest fragmentem kodu, który możemy wielokrotnie wykorzystywać w różnych miejscach programu. Funkcja pełni więc rolę podprogramu. Programowanie oparte o funkcje jest cechą pewnego paradygmatu (metodologii) programowania, która nazywana jest programowaniem proceduralnym lub strukturalnym. Z tego punktu widzenia cały program komputerowy staje się zespołem współpracujacych ze sobą podprogramów (funkcji), zaś funkcja `main()` tak naprawdę zaczyna służyć do zarządzania nimi. Ponadto raz napisaną funkcję można wykorzystywać także w innych programach.

Użycie własnej funkcji w programie wymaga jej zdefiniowania lub zadeklarowania przed użyciem funkcji `main()`. Kompilator analizuje bowiem treść programu po kolei, od pierwszego do ostatniego wiersza. Zadeklarowanie oznacza, że funkcja `main()` jest poinformowana o użyciu w jej wnętrzu własnej funkcji użytkownika, której treść (ciało) będzie znajdowała się po zamknięciu nawiasu klamrowego funkcji `main()`. Podobna zasada dotyczy zresztą używania zmiennych. Typ funkcji nie może być typem tablicowym ani funkcyjnym, ale może być wskaźnikiem do jednego z nich. Instrukcje w bloku (ciele) funkcji mogą być instrukcjami deklaracji. Ostatnią instrukcją przed nawiasem klamrowym, zamykającym ciało funkcji, musi być instrukcja `return`. Wyjątkiem jest funkcja `void`, dla której instrukcja `return` jest opcją.

Ogólna deklaracja funkcji jest następująca:

```
typ_wyniku nazwa_funkcji (deklaracja_argumentów) //nagłówek funkcji
{
  polecenia do wykonania; //treść (ciało) funkcji
}
```

Wywołanie funkcji jest np. poleceniem obliczenia wartości wyrażenia, zwracanemu przez nazwę funkcji. Jego składnia jest następująca:

```
nazwa (argumenty_aktualne)
```

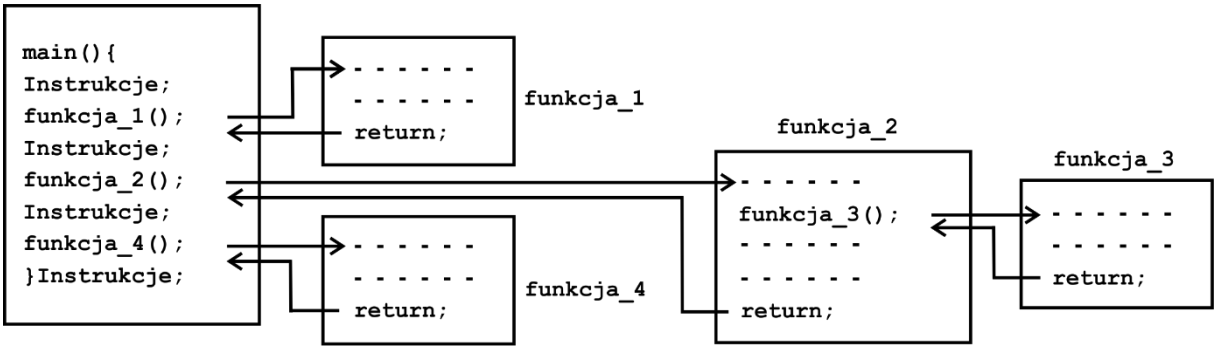
gdzie: `nazwa` jest zadeklarowaną wcześniej nazwą funkcji, natomiast `argumenty_aktualne` są po prostu wartościami `argumentów formalnych`.

Funkcje wykorzystywane są często do powtarzających się obliczeń. W tym wypadku funkcja zwraca wyliczoną wartość wyrażenia za pomocą polecenia:

```
return wyrażenie;
```

Na marginesie, podobnie działają funkcje matematyczne. Zadaniem funkcji $f(x) = x^2$, jest obliczenie wartości wyrażenia, pod warunkiem otrzymania przez funkcję odpowiedniego argumentu (parametru) x. Dla np. argumentu (parametru) $x = 3$ funkcja zwraca $f(3) = 9$.

Funkcja niekoniecznie musi zwracać jakąś wartość. Może na przykład wykonać tylko jakąś operację, bez zwracania wyniku, nie żądając przy tym żadnych argumentów (parametrów). W takim wypadku powinna być zadeklarowana jako typu `void` i określamy ją mianem `procedury`. Nic też nie stoi na przeszkodzie, by z wnętrza jednej funkcji wywoływać kolejną, jak to przedstawiono na rysunku poniżej.



Przykład 1.
Zapoznaj się z deklaracją funkcji `iloczyn`, obliczającej wynik mnożenia dwóch liczb całkowitych.

```
int iloczyn (int x, int y)
{
  return x * y;
}
```

Przykład 2.
Przeanalizuj program ilustrujący dwa sposoby użycia własnej funkcji, której zadaniem jest jedynie zatrzymanie działania programu do czasu naciśnięcia dowolnego klawisza. Zauważ, że funkcja nie zwraca żadnej wartości. W związku z tym nie zawiera instrukcji `return`.

Listing 1:
```
#include <iostream>
#include <conio.h>

using namespace std;

void dowolny(void) //deklaracja i cialo funkcji
{
  cout << "Nacisnij dowolny klawisz...";
  getch();
}

int main()
{
  dowolny();
}
```

Listing 2:
```
#include <iostream>
#include <conio.h>

using namespace std;

void dowolny(void); //sama deklaracja uzycia funkcji

int main()
{
  dowolny();
}

void dowolny(void) // cialo funkcji
{
  cout << "Nacisnij dowolny klawisz...";
  getch();
}
```

Uwaga:

1. W pierwszym listingu całą funkcję dowolny zdefiniowano przed pierwszym użyciem funkcji `main()`. W drugim listingu treść funkcji dowolny znalazła się dopiero po użyciu funkcji `main()`, ale sama funkcja musi być zadeklarowana przed nią (ze średnikiem na końcu!). W przeciwnym wypadku kompilator zgłosi błąd.
2. Oba sposoby użycia własnej funkcji są zatem równoważne i do programisty należy wybór któregokolwiek z nich. W celu zwiększenia czytelności kodu źródłowego zaleca się, aby w wypadku definiowania dużej liczby własnych funkcji umieszczać przed funkcją `main()` same deklaracje, a ich treść po funkcji głównej.

Przykład 3.
Przeanalizuj trzy programy ilustrujące różne sposoby sprawdzenia, czy dana liczba jest liczbą pierwszą. Pierwszy sposób – bez użycia funkcji, drugi z definicją funkcji, a trzeci z deklaracją funkcji przed jej użyciem.

Listing_1:
```
//Sprawdzenie pierwszosci liczb

#include <iostream>
#include <cstdlib>
#include <cmath>

using namespace std;

int main()
{
    int candidate;
    bool prime = true;

    cout << "Podaj kandydata (>= 2) na liczbe pierwsza: ";
    cin >> candidate;

        for (int i = 2; i <= sqrt(candidate); i++)
            if (candidate % i == 0) {prime = false; break;}

    if (candidate < 2) cout << "Za mala liczba!" << endl;
    else if (prime) cout << "Jest to liczba pierwsza" << endl;
    else cout << "Jest to liczba zlozona" << endl;
    return 0;
}
```

Listing_2:
```
//Sprawdzenie pierwszosci liczb
#include <iostream>
#include <cstdlib>
#include <cmath>

using namespace std;

bool prime (int candidate) //definicja (deklaracja i cialo) funkcji
{
    for (int i = 2; i <= sqrt(candidate); i++)
        if (candidate % i == 0) return false;
    return true;
}

int main()
{
    int candidate;
    cout << "Podaj kandydata (>= 2) na liczbe pierwsza: ";
    cin >> candidate;
        if (candidate < 2) cout << "Za mala liczba!" << endl;
        else if (prime(candidate)) cout << "Jest to liczba pierwsza" <<
endl;
        else cout << "Jest to liczba zlozona" << endl;
    return 0;
}
```

Listing_3:
```
//Sprawdzenie pierwszosci liczb
#include <iostream>
#include <cstdlib>
```

```
#include <cmath>

using namespace std;

bool prime (int candidate); //sama deklaracja uzycia funkcji

int main()
{
    int candidate;
    cout << "Podaj kandydata (>= 2) na liczbe pierwsza: ";
    cin >> candidate;
        if (candidate < 2) cout << "Za mala liczba!" << endl;
        else if (prime(candidate)) cout << "Jest to liczba pierwsza" <<
endl;
        else cout << "Jest to liczba zlozona" << endl;
    return 0;
}

//cialo funkcji prime
bool prime (int candidate)
{
    for (int i = 2; i <= sqrt(candidate); i++)
        if (candidate % i == 0) return false;
    return true;
}
```

Wywołanie danej funkcji zawiesza wykonanie funkcji wywołującej (która jest dla niej funkcją główną) i powoduje zapamiętanie adresu następnej instrukcji do wykonania po powrocie z funkcji wywołującej. Adres ten, zwany adresem powrotnym, zostaje umieszczony we fragmencie pamięci operacyjnej zwanej stosem programu (ang. *run-time stack*).

Argumenty do funkcji można przekazywać na trzy sposoby: przez wartości, przez wskaźniki i przez referencje. Przekazywanie argumentów przez wartość jest domyślnym mechanizmem stosowanym w języku C++. Stosowane jest wówczas, gdy nie chcemy, by wartość parametrów aktualnych była zmieniana wewnątrz funkcji lub gdy w miejsce parametrów chcemy wstawić wartości liczbowe (jak w Przykładzie 3). Nie zawsze jest to korzystne. Dla przykładu, zbudujmy funkcję zamieniającą miejscami wartości zmiennych `x` i `y`.

```
void zamiana (int x, int y)
{
  int temp = y;
  y = x;
  x = temp;
}
```

Wartości zmiennych `x` i `y` faktycznie zostaną zamienione, lecz tylko lokalnie, w ramach funkcji zamieniającej i w żaden sposób nie będzie można ich odesłać do funkcji wywołującej funkcję `zamiana`. Zadanie to można wykonać albo przez zastosowanie wskaźników, albo przez referencję. Najpierw zobaczmy mniej czytelną wersję przekazania argumentów (parametrów) przez wskaźniki.

```
#include <iostream>

using namespace std;

void zamiana (int* x, int* y)
{
  int temp = *y;
  *y = *x;
  *x = temp;
}
```

```
int main()
{
    int i = 1, j = 2;
    cout << "Przed zamiana i = " << i << ", natomiast j = " << j << endl;
    zamiana (&i, &j);
    cout << "W wyniku zamiany i = " << i << ", natomiast j = " << j <<
endl;
    return 0;
}
```

Znacznie czytelniejsze jest rozwiązanie tego samego zadania poprzez przekazanie argumentów do funkcji przez referencje, która mają dostęp do oryginalnych wartości zmiennych `x` i `y`, ponieważ znają ich adresy w pamięci.

```
#include <iostream>

using namespace std;

void zamiana (int& x, int& y)
{
  int temp = y;
  y = x;
  x = temp;
}

int main()
{
    int i = 1, j = 2;
    cout << "Przed zamiana i = " << i << ", natomiast j = " << j << endl;
    zamiana (i, j);
    cout << "W wyniku zamiany i = " << i << ", natomiast j = " << j <<
endl;
    return 0;
}
```

Każda funkcja powinna mieć unikatową nazwę. Jest to zasadne, jeśli różne funkcje mają wykonywać różne operacje. Jeśli jednak mają wykonywać podobne operacje, tyle że na różnych obiektach, wówczas korzystne jest nadanie funkcjom takiej samej nazwy. Składnia języka C++ dopuszcza taką sytuację, która nazywana jest **przeciążaniem (przeładowywaniem) funkcji**. Obowiązują jednak pewne zasady:

1. Funkcje przeciążone muszą mieć taki sam zasięg (np. bloku instrukcji, plik, programu).
2. Funkcje, które różnią się tylko zwracanym typem, nie mogą mieć takiej samej nazwy.
3. Funkcje o takiej samej nazwie muszą różnić się sygnaturami oraz mogą różnić się typem zwracanym.
4. Jeśli argumenty dwóch funkcji różnią się tylko tym, że jedna ma np. argument typu `T`, a druga typu `T&`, to funkcje te nie mogą mieć takiej samej nazwy.

Przyjrzyjmy się przykładowi, w którym zadeklarowano trzy funkcje o takich samych nazwach, ale różniące się liczbą parametrów.

```
#include <iostream>
using namespace std;

int obliczSume(int a)
{
  return a;
}

int obliczSume(int a, int b)
{
  return a + b;
}
```

```
int obliczSume(int a, int b, int c)
{
  return a + b + c;
}

main()
{
 int x = 1, y = 2, z = 3;
 cout << obliczSume(x, y) << endl;
 cout << obliczSume(x) << endl;
 cout << obliczSume(z, x) << endl;
 cout << obliczSume(x, y, z) << endl;
 cout << obliczSume(y)<<endl;
 return 0;
}
```

Programu wypisze tutaj ciąg liczb: 3, 1, 4, 6, 2. W zależności bowiem od liczby argumentów (parametrów), użyta zostanie któraś z zadeklarowanych funkcji (pierwsza dla jednego przekazanego argumentu, druga dla dwóch oraz trzecia dla trzech przekazanych argumentów).

Iteracja i rekurencja w języku C++

Pojęcie iteracji *(ang. iteration – powtarzanie)* można utożsamiać z obiegiem pętli. W każdym obiegu algorytm powtarza bowiem te same czynności.

Natomiast rekurencja *(ang. recursion – rekurencja, rekursja)* oznacza zdolność podprogramu (funkcji, procedury) do wywołania samego siebie. Muszą tu jednak być spełnione dwa warunki: jest określony jeden (lub więcej) początkowych wyrazów ciągu oraz kolejne wyrazy ciągu obliczane są za pomocą poprzednich. Na ogół też algorytmów rekurencyjnych – w przeciwieństwie do iteracyjnych – nie przedstawia się za pomocą schematów blokowych. Stosuje się za to opis słowny lub listę kroków.

Istnieje wiele algorytmów, które mogą być realizowane w sposób iteracyjny lub rekurencyjny, chociaż nie zawsze iterację można zastąpić rekurencją. Procedury lub funkcje rekurencyjne, które można zastąpić procedurami lub funkcjami iteracyjnymi, nazywamy pseudorekurencyjnymi.

Klasycznym przykładem porównania iteracji i rekurencji jest definicja silni. Definicja iteracyjna ma postać:

$$n! = \begin{cases} 1 & dla\ n = 0 \\ 1 \cdot 2 \cdot 3 \cdots \cdot n & dla\ n > 0 \end{cases}$$

Program w języku C++, realizujący ten algorytm, może mieć postać:

```
#include <iostream>

using namespace std;

long long int silnia (int n)
{
    long long int silnia = 1;
    for (int i = 1; i <= n; i++)
        silnia = silnia * i;
    return silnia;
}

int main()
{
    int n;
    cout << "Podaj n >= 0: ";
```

```
    cin >> n;
    cout << n << "! = " << silnia(n) << endl;
    return 0;
}
```

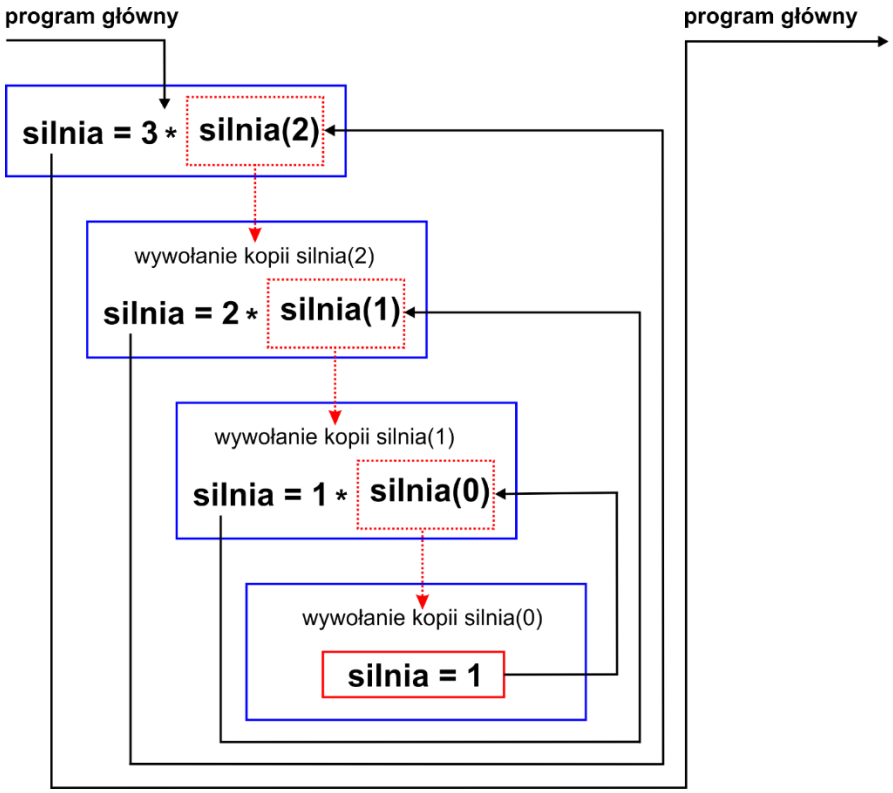
Definicja rekurencyjna silni jest natomiast następująca:

$$n! = \begin{cases} 1 & dla\ n = 0 \\ n \cdot (n-1)! & dla\ n > 0 \end{cases}$$

A sama funkcja, odpowiadająca tej definicji, może mieć wygląd:

```
long long int silnia (int n)
{
    if (n == 0) return 1;
    else return n * silnia(n - 1);
}
```

Działanie pętli iteracyjnych znane jest z wcześniejszych lekcji, pozostaje zatem wyjaśnienie działania rekurencji.



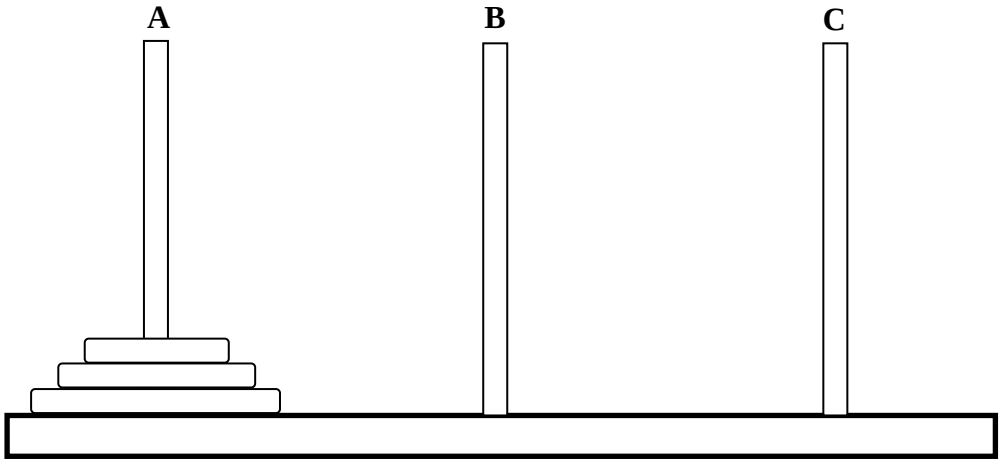
Wywołanie tej funkcji dla $n = 3$ ma następujący przebieg:

1. Program główny wywołuje funkcję $silnia(3)$. Następnie ma miejsce zapamiętanie tzw. adresu powrotnego do programu głównego, a dalej sprawdzenie, czy $n = 0$, ponieważ $0! = 1$ jest jedyną zdefiniowaną na stałe wartością. Czynności te odbywają się na tzw. pierwszym poziomie rekurencji.
2. Ponieważ $n > 0$, następuje chwilowe zawieszenie obliczeń i zapamiętanie adresu powrotnego do funkcji $silnia(3)$, gdyż to właśnie ona próbuje wykonać swoją „kopię” $silnia(2)$. Funkcja $silnia$ zeszła zatem w głąb – na drugi poziom rekurencji.
3. Wywołanie funkcji $silnia(2)$ powoduje te same skutki, co wywołanie funkcji $silnia(3)$. Podobnie rzecz się ma z wywołaniem funkcji $silnia(1)$, gdyż do jej wykonania też potrzebna jest „kopia” $silnia(0)$.
4. Dopiero na czwartym poziomie rekurencji następuje wywołanie wykonywalnej „kopii” funkcji $silnia(0)$, która z definicji ma przypisaną wartość równą 1.

5. Teraz możliwe jest wykonanie funkcji $silnia(1)$ oraz nadanie jej odpowiedniej wartości, a po niej kolejno do funkcji $silnia(2)$ oraz $silnia(3)$. Następuje więc powrót w górę – na coraz wyższe poziomy rekurencji.
6. Powrót na poziom pierwszy i wykonanie funkcji $silnia(3)$ powoduje w następstwie przekazanie sterowania do programu głównego. Funkcja $silnia(3)$ otrzymuje więc wartość 6, która zostanie wykorzystana w programie głównym.

Do wykonania rekurencyjnego podprogramu wykorzystuje się tzw. stos rekursji, na którym układa się kolejne „kopie" wraz z adresami powrotu. Zdejmowanie ze stosu następuje w odwrotnej kolejności. W informatyce taki stos nazywa się też kolejką typu LIFO *(ang. **L**ast **I**n **F**irst **O**ut – ostatni wszedł, pierwszy wyszedł)*. Należy również uzupełnić, że nie wszystkie języki programowania (np. BASIC, Cobol, Fortran) umożliwiają korzystanie z rekurencji.
Ciekawym przykładem porównania iteracji i rekurencji jest też starożytny problem tzw. wież Hanoi *(ang. towers of Hanoi)*. Jest to łamigłówka złożona z trzech pionowych pałeczek *A, B, C* i różnej wielkości krążków, które nanizano na pierwszą z nich w ten sposób, że średnice krążków rosną ku podstawie.

Zadanie polega na przeniesieniu n krążków (w tym przykładzie trzech) z pałeczki A na pałeczkę C przy ograniczeniu, że w jednym kroku można przenieść tylko jeden krążek i nie wolno kłaść krążka większego na mniejszy. Pałeczka B pełni rolę pomocniczą.



Analizę problemu należy rozpocząć od spostrzeżenia, że gdyby $n = 1$, wówczas do rozwiązania wystarczy jeden krok: $A \longrightarrow C$.

Dla $n = 2$ konieczne są trzy kroki:

$$A \longrightarrow B;\ A \longrightarrow C;\ B \longrightarrow C.$$

Dla $n = 3$ wymaganych jest już siedem kroków:

$$A \longrightarrow C;\ A \longrightarrow B;\ C \longrightarrow B;\ A \longrightarrow C;\ B \longrightarrow A;\ B \longrightarrow C;\ A \longrightarrow C.$$

Łatwo zauważyć, że liczba kroków zmienia się zgodnie z zależnością $2^n - 1$. Legenda głosi, że w pewnym tybetańskim klasztorze mnisi buddyjscy od 3000 lat przenoszą 64 złote krążki w tempie jeden krążek na sekundę. Z chwilą przeniesienia ostatniego krążka nastąpi koniec świata. Nie trzeba się jednak tym zbytnio przejmować, albowiem $(2^{64} - 1)$ sekund odpowiada mniej więcej 585 mld lat (wiek Ziemi szacuje się na ok. 5 mld lat). Równanie rekurencyjne dla tego problemu jest następujące:

$$H(n) = 2H(n-1) + 1$$

przy warunku $H(1) = 1$.

Stosując metodę indukcji zupełnej można udowodnić, że $H(n) = 2^n - 1$.

Przechodząc do konstrukcji rekurencyjnego algorytmu Hanoi (dla $n > 1$) trzeba zauważyć, że zawiera on trzy etapy:

1. Przenieś $n-1$ górnych krążków z pałeczki A na pałeczkę B, używając C.
2. Przenieś największy krążek z pałeczki A na pałeczkę C.
3. Przenieś wszystkie krążki z pałeczki B na pałeczkę C używając A.

Zatem opis słowny algorytmu (n, A, C, B) może być następujący:

1. Jeśli $n = 1$, to $(1, A, C)$ i zakończ algorytm.
2. Jeśli $n > 1$, to zastosuj algorytm dla $(n-1, A, B, C)$.
3. Przenieś pozostały krążek $(A \longrightarrow C)$.
4. Zastosuj ten algorytm dla $(n-1, A, C, B)$.

Istnieje również iteracyjne rozwiązanie problemu wież Hanoi, dokładnie równoważne wersji rekurencyjnej. Wpierw jednak należy sobie wyobrazić, że pałeczki ustawione są na obwodzie koła.

Algorytm iteracyjny dla $n > 1$ działa następująco:

1. Poruszając się w kierunku przeciwnym do ruchu wskazówek zegara przenieś najmniejszy krążek z pałeczki, na której akurat spoczywa, na następny.
2. Wykonaj jedyne możliwe przeniesienie, nie dotyczące najmniejszego krążka.

Algorytm iteracyjny jest niezwykle łatwy do wykonania. Można udowodnić, że daje on dokładnie taką samą sekwencję przeniesień, co odmiana rekurencyjna.

Strumienie i pliki

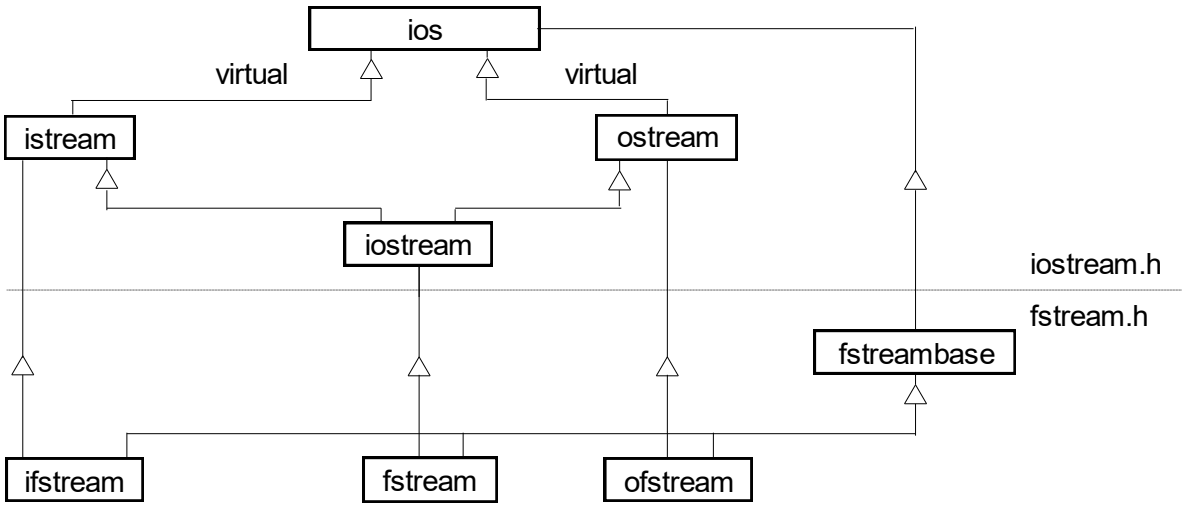
Strumień jest typem abstrakcyjnym, opisującym urządzenie logiczne, które pobiera lub wysyła informację w sposób sekwencyjny. Oznacza to, że dostęp do $n - tej$ wartości w strumieniu danych jest możliwy po uzyskaniu dostępu do poprzednich $(n-1)$ wartości.

W języku C++ wszystkie strumienie zachowują się w ten sam sposób, co pozwala na dołączanie ich do urządzeń fizycznych o różnych własnościach. Tak więc można wykorzystać tę samą metodę do wyprowadzenia informacji na ekran, drukarkę lub do pliku dyskowego (tzw. polimorfizm).

Uruchomienie programu w języku C++ powoduje automatyczne otwarcie czterech strumieni, których deklaracje zawarte są w pliku **`iostream`** i w związku z tym plik ten musi być dołączony.

Aby korzystać ze strumieniowych operacji plikowych, należy zadeklarować bibliotekę **`fstream.`** Biblioteka ta zawiera definicje klas dających prosty dostęp do plików. Pliki te traktowane są jak strumienie wejściowe lub wyjściowe, reprezentowane przez obiekty następujących klas:

- `ifstream` – pliki wejściowe,
- `ofstream` – pliki wyjściowe,
- `fstream` – pliki wejściowo-wyjściowe.



Obiekty reprezentujące pliki mogą być argumentami operatorów strumieniowych, czyli << oraz >>. Aby korzystać z funkcji wejścia/wyjścia, najpierw należy utworzyć obiekt odpowiedniej klasy (np. `ofstream`), a następnie korzystać z niego tak, jak ze standardowego strumienia. Wymienione klasy udostępniają funkcje działające tylko na plikach:

- `open` – otwarcie pliku,
- `close` – zamknięcie pliku.

Aby utworzyć np. plik `info.txt` do odczytu, należy podać następujące polecenia:

`ifstream dane;` //tworzy obiekt `dane` klasy `ifstream`, który staje się identyfikatorem strumienia wejściowego,
`dane.open("info.txt");` //otwiera do odczytu plik `info.txt` znajdujący się w katalogu bieżącym dysku bieżącego.

Jeśli plik znajduje się w innym miejscu, należy wpisać pełną ścieżkę dostępu, stosując jednak znak specjalny `\\` np. `C:\\dane\\info.txt`.

Zadanie 1.
Napisać program tworzący plik `dane.txt` zawierający liczby.

Rozwiązanie:

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

using namespace std;

main()
{
 ofstream plik;   //utworzenie obiektu plik klasy ofstream
 plik.open("dane.txt");   //otwarcie pliku dane.txt do zapisu

 for (int i=0; i<9; i++)
 plik<<i<<endl;  //zapis do pliku wartosci zmiennej i
 plik.close();   //zamkniecie pliku dane.txt
 plik.open("dane.txt");   //otwarcie pliku dane.txt do zapisu
 int a=5, b=4, c=3;
 plik<<a<<"\n"<<b<<endl<<c<<endl;   //zapis do pliku zmiennych a, b, c
 plik<<endl<<++a;   //zwiekszenie a o 1 i zapis a do pliku
 plik.close();   //zamkniecie pliku dane.txt
```

```
getch();
return 0;
}
```

Pliki mogą być otwierane w różnych trybach za pomocą tzw. specyfikatorów:

`ios::app` – dopisywanie danych na końcu pliku,
`ios::ate` – przesunięcie aktualnej pozycji w pliku na koniec pliku,
`ios:in` – otwarcie pliku do odczytu (tryb domyślny dla klasy `ifstream`),
`ios:out` – otwarcie pliku do zapisu (tryb domyślny dla klasy `ofstream`),
`ios:binary` – otwarcie pliku w trybie binarnym (domyślnie w trybie tekstowym),
`ios::trunc` – zapisywanie na dotychczasowej zawartości pliku,
`ios::nocreate` – plik zostanie otwarty pod warunkiem, że już istnieje, czyli nie tworzy nowego pliku,
`ios:noreplace` – plik zostanie otwarty, jeśli dotychczas nie istniał, czyli zostanie utworzony nowy plik.

Przykłady:

1. Otwarcie pliku do odczytu (domyślnie `ios::in`):

```
ifstream baza;
baza.open("baza.txt");
```

2. Otwarcie pliku do odczytu, jeśli plik istnieje:

```
ifstream archiwum;
archiwum.open("arch.txt", ios::nocreate);
```

3. Otwarcie pliku do zapisu (domyślnie `ios::out`) na końcu pliku w trybie binarnym:

```
ofstream archiwum;
raport.open("raport", ios::binary & ios::app);
```

Zadanie 2.
Utwórz w Notatniku plik `dane1.txt` i umieść w nim następujące słowa (każde w nowym wierszu): absolutny, adapter, adnotacja, adres, bariera, basen, bat, bateria, baza, beczka, bestia, bez, cel, celny, cena. Następnie napisz program, który wczyta słowa z pliku `dane1.txt` i przepisze do pliku `wyniki1.txt` tylko te słowa, które mają parzystą liczbę znaków.

Rozwiązanie:

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

using namespace std;

main ()
{
 ifstream fin("dane1.txt");
 ofstream fout("wyniki1.txt");
 string s;
 while (!fin.eof())
 {
  fin>>s;
  if (s.size()%2==0) fout<<s<<"\n";
 }
 fin.close();
 fout.close();
getch();
```

```
return 0;
}
```

Zadanie 3.
Utwórz w Notatniku plik `dane2.txt` i umieść w nim 10 par liczb (każda para w nowym wierszu, a liczby rozdzielone spacją) z przedziału od 5 do 500. Następnie napisz program, który mnoży każdą parę przez siebie i przepisuje do pliku `wyniki2.txt` iloczyny tylko tych par liczb, które zawarte są w przedziale od 250 do 3000.

Rozwiązanie:
```
#include <iostream>
#include <fstream>
#include <conio.h>

using namespace std;

main ()
{
 ifstream fin("dane2.txt");
 ofstream fout("wyniki2.txt");
 int liczba1, liczba2, iloczyn;
 while (!fin.eof())
 {
  fin>>liczba1>>liczba2;
  iloczyn=liczba1*liczba2;
  if (iloczyn>=250&&iloczyn<=3000)
   fout<<iloczyn<<"\n";
 }
 fin.close();
 fout.close();
 getch();
 return 0;
}
```

Operacje na napisach w języku C++

Napisem lub łańcuchem tekstowym nazywamy ciąg znaków zakończony znacznikiem `'\0'`, który sygnalizuje koniec tekstu. Napis może być przechowywany w tablicach jednowymiarowych lub jako zmienna łańcuchowa klasy `cstring`.

Deklaracja tablicy znakowej może mieć postać:

```
char identyfikator[rozmiar];
```

Należy pamiętać o tym, że znaki w takiej tablicy numerowane są od zera oraz że rozmiar tablicy musi uwzględniać znacznik końca tekstu `'\0'`.

Przykłady poprawnych deklaracji tablicy znakowej

```
char tab[15]={"informatyka"};
char tab[15]= "informatyka";
char tab[15]={ 'i','n','f','o','r','m','a','t','y','k','a','\0'};
```

Wszystkie powyższe deklaracje są równoważne. Jeśli rozmiar deklarowanej tablicy jest większy niż liczba wpisanych znaków, pozostałe elementy przyjmują wartość zero. Podczas deklaracji rozmiar tablicy nie musi być podany, ponieważ w takiej sytuacji zostanie on określony automatycznie na podstawie wartości początkowej, np.

```
char tab[]={"informatyka"};
```

Język C++ zawiera wiele przydatnych funkcji operujących na tablicach znakowych.

Funkcja	Opis	Przykład użycia (char s[]="abcdef";)	Wynik działania
char *strcpy (char *s1, const char *s2)	Kopiowanie łańcucha s2 do łańcucha s1 (znak końca łańcucha s2 również jest kopiowany). Wartością funkcji jest s1.	strcpy(s, "nowy");	s="nowy"
char *strncpy (char *s1, const char *s2, int n)	Kopiowanie n znaków z łańcucha s2 do łańcucha s1. Wartością funkcji jest s1.	strncpy(s, "nowy", 3);	s="nowdef"
char *strcat (char *s1, const char *s2)	Dołączanie (konkatenacja) łańcucha s2 na koniec łańcucha s1. Wartością funkcji jest łańcuch utworzony w wyniku połączenia.	strcat(s, "nowy");	s="abcdefnowy"
int strlen (const char *s)	Wyznaczenie liczby znaków w łańcuchu s.	int a=strlen(s);	6
int strcmp (const char *s1, const char *s2)	Porównanie łańcuchów s1 i s2. Jeśli s1<s2 (czyli s1 jest alfabetycznie przed s2), to wartość funkcji jest mniejsza od zera, jeśli s1=s2, to jest równa zeru, a jeśli s1>s2 (czyli s1 jest alfabetycznie po s2), to wartość funkcji jest większa od zera.	int a=strcmp(s, "abcdef");	0

Przykład 1.
Przeanalizuj kod źródłowy, który w tekście wprowadzonym przez użytkownika wykonuje zamianę znaków "a" na "*", a następnie znaków różnych od "m" i "n" na "U".

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

using namespace std;

void wczytaj (char s[])
{
 cout<<"podaj tekst: ";
 cin.getline(s,256); //wczytywanie maksymalnie 256 znaków z klawiatury
}

char *zamien1 (char s[])
{
 int dl=strlen(s);
 for (int i=0;i<dl;i++)
  if (s[i]=='a') s[i]='*';
 return s;
}

char *zamien2 (char s[])
{
 int dl=strlen(s);
```

```
 for (int i=0;i<dl;i++)
  if (s[i]!='m'&&s[i]!='n') s[i]='U';
 return s;
}

main()
{
 char s[256];
 wczytaj(s);
 cout<<"\ntekst po zamianie 1: "<<zamien1(s)<<endl;
 cout<<"tekst po zamianie 2: "<<zamien2(s)<<endl;

system("pause");
 return 0;
}
```

Napisy można również przechowywać korzystając z narzędzi klasy `cstring`. Aby tego dokonać, należy zadeklarować użycie odpowiedniej biblioteki:

```
#include <cstring>
```

Klasa `cstring` oferuje dostęp do wielu operatorów do wykonywania operacji na łańcuchach, przykładowo:

- operator przypisania: `=`,
- operatory relacyjne: `==`, `!=`,
- operator konkatenacji, czyli połączenia łańcuchów: `+`,
- operator indeksowy: `[]`.

Przykładowe sposoby deklaracji i inicjalizacji zmiennej typu string są następujące:

Deklaracja lub inicjalizacja	Opis	Wynik działania
`string s;` `s="informatyka";`	Deklaracja zmiennej `s` i przypisanie wartości zmiennej `s` (tablica znaków wymaga użycia funkcji `strcpy()`)	`s="informatyka"`
`string s("informatyka");`	Inicjalizacja zmiennej `s`	`s="informatyka"`
`string s="informatyka";`	Inicjalizacja zmiennej `s`	`s="informatyka"`
`string s(8, '*');`	Inicjalizacja zmiennej `s`	`s="********"`

Przykład 2.
Przeanalizuj poniższy listing, który ilustruje działanie operatorów na łańcuchach.

```
#include <iostream>
#include <cstring>
#include <windows.h>

using namespace std;

main()
{
 string s1, s2("Ala"), s3="Ola";

 s1="Ala";

if (s1==s2) cout<<"teksty "<<s1<<" i "<<s2<<" sa takie same"<<endl;
 else if (s1<s2) cout<<"tekst "<<s1<<" jest wczesniej w slowniku niz tekst
"<<s2<<endl;
```

```
  else cout<<"tekst "<<s1<<" nie jest pozniej w slowniku niz tekst
"<<s2<<endl;

 s2+='n';

 if (s1==s2) cout<<"teksty "<<s1<<" i "<<s2<<" sa takie same"<<endl;
 else if (s1<s2) cout<<"tekst "<<s1<<" jest wczesniej w slowniku niz tekst
"<<s2<<endl;
  else cout<<"tekst "<<s1<<" jest pozniej w slowniku niz tekst "<<s2<<endl;

 s3+="!!!";

 cout<<s3<<endl;

 s3[1]='*';

 cout<<s3<<endl;

 system("pause");
 return 0;
}
```

Wykonywanie operacji z wykorzystaniem typu `string` jest bardzo efektywne. W poniższej tabeli opisane są przydatne metody, których nazwy zależą od nazw zmiennych, dla jakich mają zastosowanie.

Funkcja	**Opis**	**Przykład użycia** (`string s ="abcdef";`)	**Wynik działania**
`bool empty (void)`	Zwraca wartość `true`, jeśli napis jest pusty, a `false` w przeciwnym wypadku	`bool a=s.empty();`	`a=0`
`int size (void)`	Zwraca liczbę znaków w napisie	`int a=s.size();`	`a=6`
`char at(int i)`	Zwraca znak o podanym indeksie `i` oraz zapobiega wyjściu poza przedział	`char a=s.at(3);`	`a='d'`
`void clear (void)`	Usuwa wszystkie znaki z łańcucha	`s.clear();`	`s=""`
`char* c_str(void)`	Konwertuje typ `string` na `char[]`	`char a[]=s.c_str();`	`a ="abcdef"`
`string substr (int i, int j)`	Zwraca podciąg znaków rozpoczynający się od pozycji `i` o długości `j` znaków	`string a=s.substr(2,3);`	`a ="cde"`
`int find (string b)`	Wyszukuje w napisie podciąg znaków `b`, zwraca pozycję rozpoczęcia szukanego podciągu w tekście lub wartość większą od długości przeszukiwanego napisu	`string b="cde"; int a=s.find(b);`	`a=2`

Przykłady konwersji łańcuchów:

1. Typu `string` na `char[]`

```
string s1="abcdef";
char s2[50];
s2=s1.c_str();
```

2. Typu `char[]` na `string`

```
string s1;
char s2[]"abcdef";
s1=s2;
```

Przy pisaniu programów przydatne mogą okazać się także trzy inne funkcje:

`bool isalnum(char znak)` – przyjmuje wartość `true`, jeśli znak jest literą lub cyfrą;
`bool isdigit(char znak)` – przyjmuje wartość `true`, jeśli znak jest cyfrą;
`bool isalpha(char znak)` – przyjmuje wartość `true`, jeśli znak jest literą;

Przykład użycia funkcji `isdigit`:

```
string s("ab12c3d4ef56");
 for (int i=0; i<s.size(); i++)
  if (isdigit(s[i])) cout<<s[i];
```

Po wykonaniu tego fragmentu programu wynik będzie następujący: `123456`.