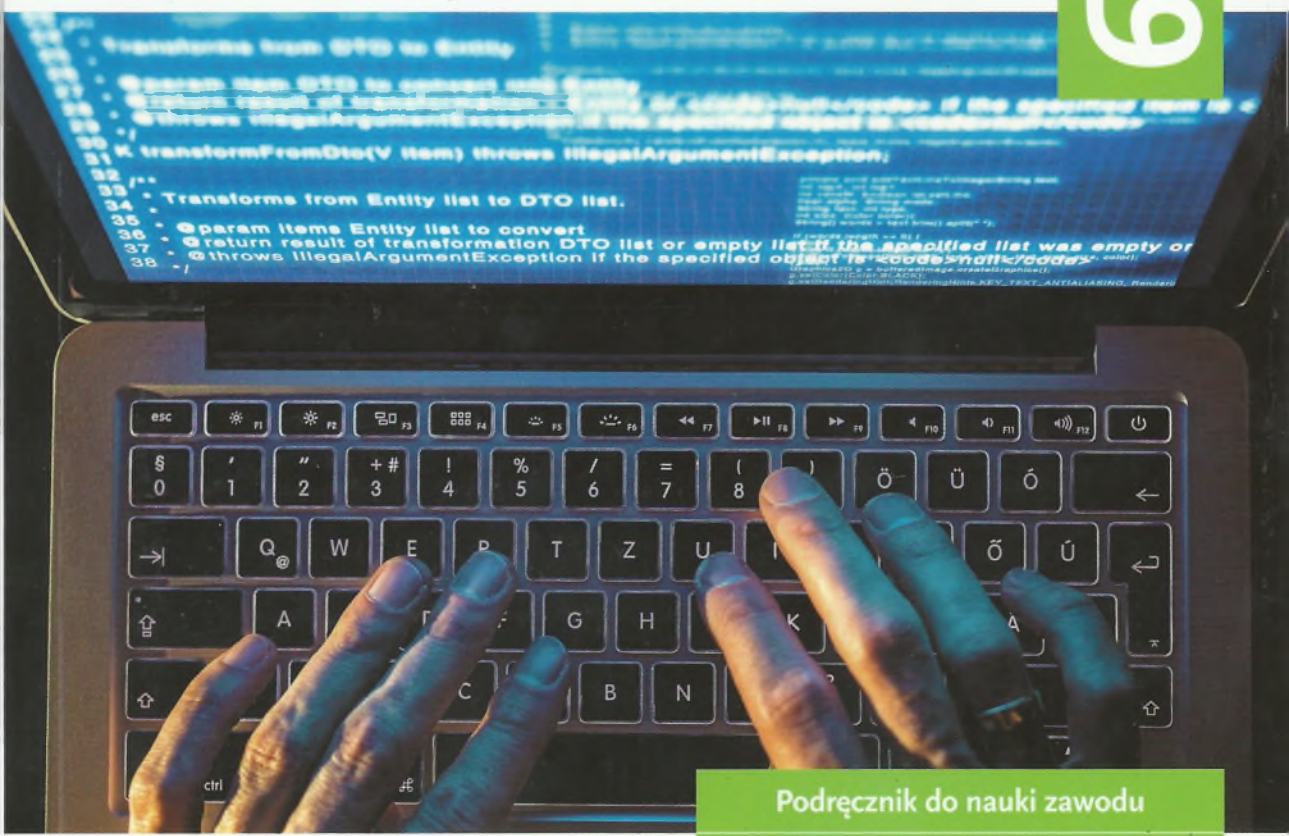
REFORMA 2017

Część 1

EE.09

Programowanie i tworzenie stron internetowych oraz baz danych i administrowanie nimi



Podręcznik do nauki zawodu

• TECHNIK INFORMATYK

WSiP

REFORMA 2017

Część 1

EE.09

Programowanie i tworzenie stron internetowych oraz baz danych i administrowanie nimi

Agnieszka Klekot
Tomasz Klekot

Podręcznik do nauki zawodu

• TECHNIK INFORMATYK

WSiP

Podręcznik dopuszczony do użytku szkolnego przez ministra właściwego do spraw oświaty i wychowania i wpisany do wykazu podręczników przeznaczonych do kształcenia w zawodach na podstawie opinii rzeczoznawców: dr Moniki Szymańskiej, mgr inż. Janiny Grobelnej i mgr Zofii Gondy-Ciupy.

Typ szkoły: **technikum, szkoła policealna.**
Zawód: **technik informatyk.**
Kwalifikacja: **EE.09. Programowanie, tworzenie i administrowanie stronami internetowymi i bazami danych.**

Rok dopuszczenia: 2018.

Wydanie I (2018)

ISBN 978-83-02-17917-4 (całość)
ISBN 978-83-02-17361-5 (część 1)

Opracowanie merytoryczne i redakcyjne: **Zbigniew Dziedzic** (redaktor prowadzący)
Konsultacja: **dr inż. Krzysztof Pytel**
Redakcja językowa: **Izabela Majewska**
Redakcja techniczna: **Elżbieta Walczak**
Projekt okładki: **Dominik Krajewski**
Fotoedycja: **Agata Bażyńska-Khan**
Skład i łamanie: **ALINEA Ewa J. Kamińska**
Fotografia na okładce: Martchan/Shutterstock.com

Wydawnictwa Szkolne i Pedagogiczne Spółka Akcyjna
00-807 Warszawa, Aleje Jerozolimskie 96
KRS: 0000595068
Tel.: 22 576 25 00
Infolinia: 801 220 555
www.wsip.pl
Druk i oprawa: ArtDruk Zakład Poligraficzny Andrzej Łuniewski

SPIS TREŚCI

7. Wskaźniki

8. Operacje na plikach

9. Programowanie obiektowe

10. Projektowanie i dokumentowanie programów

1. Wstęp do programowania

- Pojęcia podstawowe
- Reprezentacje algorytmów
- Klasyfikacja algorytmów i ich przykłady

1.1 Pojęcia podstawowe

ZAGADNIENIA

- **Co to jest program?**
- **Co to jest programowanie?**
- **Strukturalny język programowania Pascal**
- **Cechy programowania obiektowego**

Program to zbiór poleceń w instrukcji zgodnych z zasadami języka programowania, których wykonanie prowadzi do zrealizowania określonego zadania.

Programowanie polega na zapisywaniu algorytmów w formie programów zrozumiałych dla komputera. Program opisuje proces przekształcania danych wejściowych w dane wyjściowe według pewnego algorytmu. Dane wejściowe muszą być dostarczone do programu przez użytkownika w celu umożliwienia wykonania algorytmu. Dane wyjściowe są generowane przez program i stanowią wyniki działania programu. Program musi być zapisany w języku programowania w postaci ciągu instrukcji, ściśle według reguł języka.

Języki programowania dzieli się przede wszystkim ze względu na stopień zaawansowania. Języki pierwszej generacji to języki maszynowe, czyli języki procesorów. Instrukcje są w nich zapisywane w postaci liczb binarnych. Ich rozkazy odpowiadają bezpośrednio instrukcjom procesora. Powstały wraz z narodzinami pierwszych komputerów.

Języki drugiej generacji, tzw. języki symboliczne, asemblery, to języki niskiego poziomu, pod względem składni tożsame z maszynowymi, z tą różnicą, że zamiast liczb używa się w nich tzw. mnemoników (mnemonik to składający się z kilku liter kod w postaci słowa, który oznacza konkretną czynność, jaką ma wykonać procesor).

Języki trzeciej generacji to języki wysokiego poziomu, proceduralne (imperatywne). Są to języki ogólnego przeznaczenia o dużym stopniu uniwersalności. W tych językach jedna instrukcja jest tłumaczona na kilka instrukcji procesora.

Pierwszym językiem tego typu był ALGOL. Do tej grupy należą m.in.: FORTH, BASIC – język niestrukturalny, Pascal, C, FORTRAN – języki strukturalne, C++, Java – języki zorientowane obiektowo.

Przykładem strukturalnego języka programowania trzeciej generacji jest Pascal. Programowanie strukturalne to programowanie wykorzystujące podział programu na moduły komunikujące się przez dobrze określone interfejsy. Jest ono rozszerzeniem koncepcji programowania proceduralnego, które zaleca dzielenie kodu na procedury wykonujące ściśle określone zadania. Procedury nie powinny korzystać z parametrów globalnych, ale przekazywać wszystkie potrzebne dane jako parametry do procedury. Programowanie strukturalne oddzielnie definiuje dane, a oddzielnie – funkcje. Jest to grupa języków, w których istnieje możliwość podzielenia programu na moduły wykonujące różne operacje wchodzące w skład programu. Programowanie strukturalne, które zrewolucjonizowało

tworzenie oprogramowania, opiera się na bardzo prostej zasadzie. Duży problem rozbija się na kilka mniejszych. Te rozbija się na jeszcze mniejsze itd. Zaletą programowania strukturalnego jest możliwość tworzenia programu zespołowo: każdy konkretny programista dostaje pewną część zadania do rozwiązania. Standardowy język Pascal nie wspiera jednak programowania obiektowego.

Następną generację języków programowania tworzą tzw. języki obiektowe – Delphi, C++, PHP, Java, Python. Programowanie obiektowe wiąże się z wykorzystaniem tzw. obiektów – elementów łączących ściśle dane i procedury. Program obiektowy korzysta z obiektów komunikujących się ze sobą w celu wykonania określonych zadań.

Cechy typowe dla programowania obiektowego:

- abstrakcja – zredukowanie właściwości opisywanego obiektu do najbardziej podstawowych;
- hermetyzacja danych – ograniczenie dostępu do składowych jest ograniczony za pomocą dobrze określonego interfejsu;
- dziedziczenie – mechanizm umożliwiający wywodzenie nowych klas z klas już istniejących, wraz z przejmowaniem ich metod;
- polimorfizm – wielopostaciowość, pozwalająca na wybór metody spośród różnych wersji w zależności od kontekstu.

Pojęcia związane z programowaniem obiektowym zostały omówione w rozdziale 9.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Zdefiniuj terminy „program" i „programowanie".
2. Podaj przykłady strukturalnego języka programowania.
3. Omów kolejne generacje języków programowych.
4. Czym różni się programowanie strukturalne od obiektowego?

1.2 Reprezentacje algorytmów

ZAGADNIENIA
- Co to jest algorytm?
- Metody reprezentacji algorytmu
- Rodzaje schematów blokowych

Algorytm określa sposób przekształcania danych wejściowych w dane wyjściowe zgodnie z wyznaczonym celem. Składa się z opisu obiektów, na których są wykonywane działania, działań realizujących cel algorytmu oraz kolejności tych działań.

Cechy algorytmu:
- poprawność,
- jednoznaczność,
- skończoność,
- efektywność.

Poprawność występuje wtedy, gdy algorytm daje oczekiwane wyniki. Jednoznaczność polega na tym, że algorytm zawsze daje te same wyniki przy takich samych danych wejściowych. Ze skończonością z kolei mamy do czynienia wówczas, kiedy algorytm wykonuje się w skończonej liczbie kroków. Natomiast efektywność związana jest z tym, że algorytm powinien prowadzić do rozwiązania zadania w jak najmniejszej liczbie kroków.

Istnieje kilka wymienionych niżej metod reprezentacji algorytmu.
1. **Słowny opis** – pierwszy opis algorytmu, który jest jego mało ścisłą reprezentacją. Rozpoczyna się dyskusją, w jaki sposób można rozwiązać dane zadanie. Służy wyrobieniu intuicji i ukierunkowaniu rozwiązań we właściwe sposoby i techniki przydatne w rozwiązaniu.
2. **Lista kroków** – dokładny sposób opisywania obliczeń i ich kolejności. Poszczególne kroki zawierają opis operacji, które mają być wykonane przez algorytm. Występują polecenia związane ze zmianą kolejności wykonania kroków lub polecenia zakończenia algorytmu.
3. **Schemat blokowy** – najpopularniejszy; składa się ze skrzynek oraz połączeń między nimi. Są tu zapisane operacje, które mają być wykonane, a połączenia wyznaczają kolejność ich wykonania.
4. **Drzewo algorytmiczne** (drzewo obliczeń) – przyjmuje postać drzewa w matematycznym tego słowa znaczeniu. W drzewie algorytmu daje się wyróżnić korzeń – wierzchołek, w którym rozpoczynają się działania algorytmu, wierzchołki pośrednie, gdzie są umieszczane operacje wykonywane w algorytmie, oraz wierzchołki końcowe (tzw. liście), odpowiadające różnym wynikom zakończenia obliczeń w algorytmie.
5. **Pseudokod** – jeśli korzystamy z tego sposobu, to rezygnujemy z reguł składniowych danego języka programowania na rzecz czytelności instrukcji.

6. **Język algorytmiczny** (język programowania) – najbardziej ścisły i zrozumiały dla komputera opis algorytmu. Algorytm zapisany w języku programowania nazywamy programem.

Najczęściej wykorzystywanym sposobem przedstawiania algorytmów jest schemat blokowy, wykorzystywany przede wszystkim przez programistów. Jest to graficzny zapis algorytmu rozwiązania danego zadania, przedstawiający opis i kolejność wykonywania czynności.

Schematy blokowe pisze się najczęściej na kartce papieru. Można jednak skorzystać z edytora tekstu MS Word czy też MS Visio. Są one wyposażone w opcje do tworzenia figur schematu, ale nie są zbyt wygodne w użyciu.

Alternatywą są programy specjalnie przeznaczone do opisywania algorytmów. Jednym z nich są Magiczne bloczki. Program ten umożliwia testowanie działania algorytmu. Jest to szczególnie przydatna funkcja dla początkujących programistów, którzy mają problemy z wyszukiwaniem błędów w programie.

W zależności od przedstawianego algorytmu stosuje się różne zestawy figur geometrycznych, zwanych blokami, których kształty reprezentują umownie rodzaje elementów składowych.

W tabeli 1.1 przedstawiono różne rodzaje bloków:

Tabela 1.1. Bloki wykorzystywane do tworzenia schematów blokowych

Schemat graficzny	Nazwa skrzynki (bloku)	Funkcja
START	skrzynka graficzna	początek algorytmu
STOP		koniec algorytmu
c=a+b;	skrzynka operacyjna	obliczanie wartości wyrażeń
Podaj n	skrzynka wejścia	wprowadzanie danych
Wypisz x	skrzynka wyjścia	wyprowadzanie danych
a > 0 TAK NIE	skrzynka warunkowa	sprawdzanie warunku

SPRAWDŹ SWOJĄ WIEDZĘ

1. Z czego składa się algorytm?
2. Wymień cechy algorytmu.
3. Opisz różne metody reprezentacji algorytmu.
4. Jakie znasz rodzaje bloków wykorzystywanych do tworzenia schematów blokowych?

1.3 Klasyfikacja algorytmów i ich przykłady

ZAGADNIENIA

- **Podział algorytmów ze względu na kolejność wykonywania działań**
- **Podział algorytmów ze względu na sposób wykonywania operacji**
- **Podział algorytmów ze względu na przeznaczenie**

Algorytmy ze względu na kolejność wykonywania działań dzielimy na sekwencyjne (proste, liniowe), z rozgałęzieniami, cykliczne (z pętlą) i mieszane.

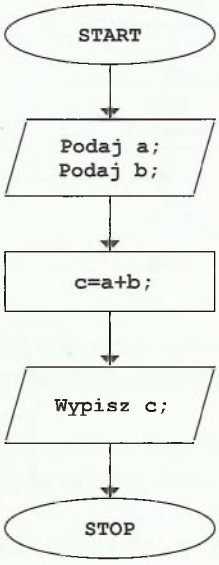
Algorytmy sekwencyjne (proste, liniowe) polegają na tym, że kolejne działania algorytmu są zawsze wykonywane w tej samej kolejności, w jakiej zostały zapisane, niezależnie od danych wejściowych; żaden krok nie może zostać pominięty ani powtórzony.

Algorytmy z rozgałęzieniami charakteryzują się tym, że kolejność wykonywania niektórych czynności może się zmienić w zależności od wartości danych wejściowych lub wartości danych pomocniczych, istnieje bowiem przynajmniej jeden blok warunkowy.

Algorytmy cykliczne (z pętlą) to takie, w których podczas wykonywania algorytmu następuje powtórzenie określonych instrukcji przy zmienionych danych wejściowych; pętla to pewna część algorytmu, która jest powtarzalna.

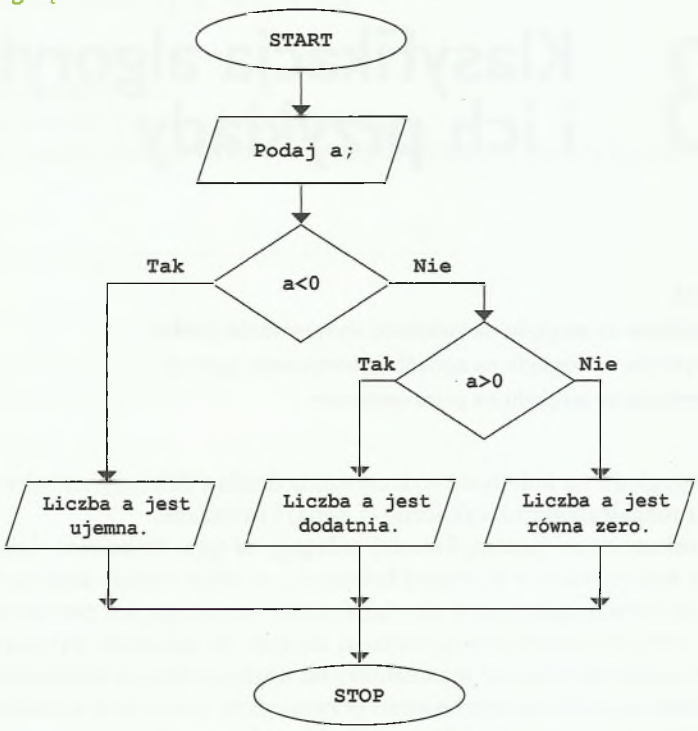
Algorytmy mieszane stanowią natomiast kombinację algorytmów opisanych powyżej.

Algorytm liniowy



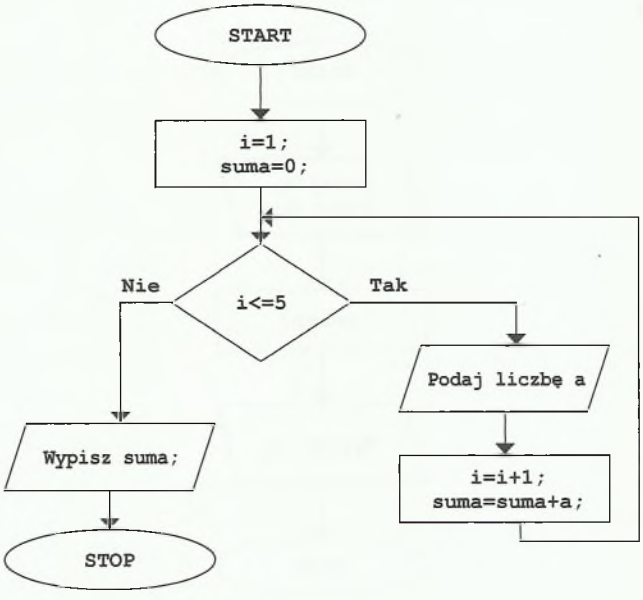
Rys. 1.1. Algorytm liniowy

Algorytm z rozgałęzieniem



Rys. 1.2. Algorytm z rozgałęzieniem

Algorytm z pętlą



Rys. 1.3. Algorytm z pętlą

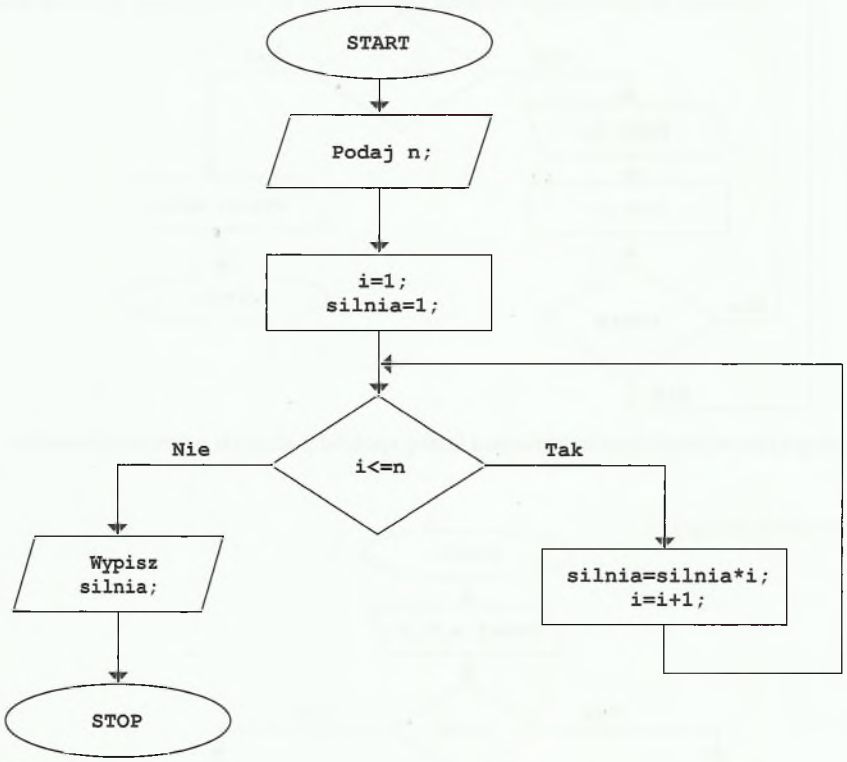
Algorytmy ze względu na sposób wykonywania operacji dzielimy na sekwencyjne, iteracyjne i rekurencyjne.

Algorytmy sekwencyjne polegają na tym, że instrukcje opisane w algorytmie są wykonywane w kolejności, w jakiej zostały wprowadzone.

Algorytmy iteracyjne charakteryzują się tym, że określone instrukcje algorytmu są powtarzane do spełnienia wymaganego warunku.

Algorytmy rekurencyjne polegają na utworzeniu formuły powtarzającej dane i odwołaniu się do niej.

Algorytm iteracyjny

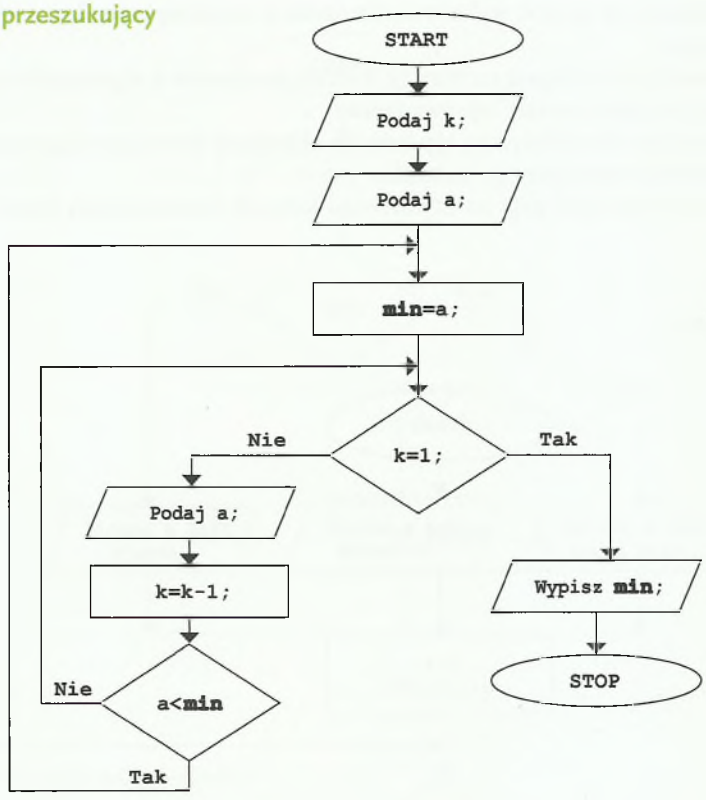


Rys. 1.4. Algorytm iteracyjny

Podział algorytmów ze względu na przeznaczenie:

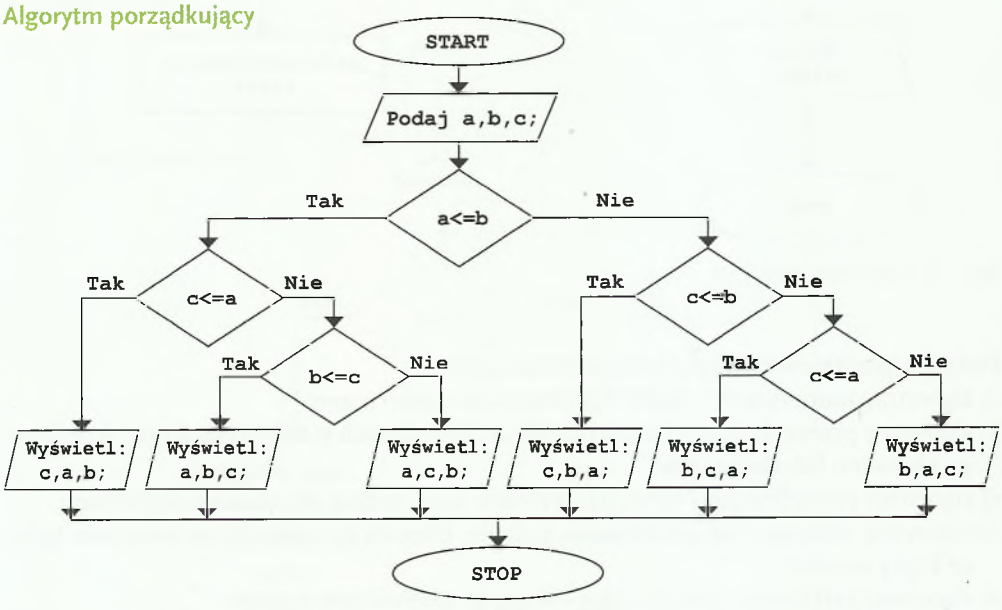
a) algorytmy numeryczne – wykonują obliczenia numeryczne;
b) algorytmy przeszukujące – badają określony zbiór danych w celu wyszukania określonego elementu lub elementów;
c) algorytmy porządkujące – ustawiają wybrane dane według określonego kryterium;
d) algorytmy rekurencyjne – rozwiązują zadanie, które da się podzielić na mniejsze, będące kopią wzorca;
e) algorytmy szyfrujące / deszyfrujące – służą do szyfrowania danych;
f) algorytmy kompresji danych – określają optymalny sposób kompresji zapisu danych.

Algorytm przeszukujący



Rys. 1.5. Algorytm wyszukujący najmniejszą liczbę spośród podanych przez użytkownika

Algorytm porządkujący



Rys. 1.6. Algorytm wyświetlający trzy liczby w postaci ciągu rosnącego

SPRAWDŹ SWOJE UMIEJĘTNOŚCI

1. Utwórz schemat blokowy algorytmu rozwiązującego dowolne równanie kwadratowe.
2. Utwórz schemat blokowy algorytmu, który obliczy średnią arytmetyczną dziesięciu liczb podanych z klawiatury.
3. Utwórz schemat blokowy algorytmu wyznaczającego największy wspólny dzielnik dwóch liczb całkowitych.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Omów podział algorytmów ze względu na przeznaczenie.
2. Omów podział algorytmów ze względu na kolejność wykonywania działań.
3. Omów podział algorytmów ze względu na sposób wykonywania operacji.

2. Podstawy języka C++

- Środowisko programistyczne
- Struktura prostego programu w C++
- Typy zmiennych w C++
- Operatory w C++
- Instrukcje wejścia/wyjścia

2.1 Środowisko programistyczne

ZAGADNIENIA

- **Program C++**
- **Struktura prostego programu w C++**
- **Komplikator**
- **Schemat komplikacji**

Jednym z najpopularniejszych języków programowania, przeznaczonych do tworzenia programów i gier, jest C++. Język ten umożliwia pisanie aplikacji dla programów operacyjnych Windows, Linuks i innych mniej popularnych systemów. Łatwość wykorzystywania możliwości sprzętowych komputera w połączeniu z tzw. przenośnością kodu źródłowego między platformami stwarza bardzo duże możliwości dla programistów na całym świecie.

Na rynku jest dostępnych kilka różnych środowisk umożliwiających programowanie w C++.

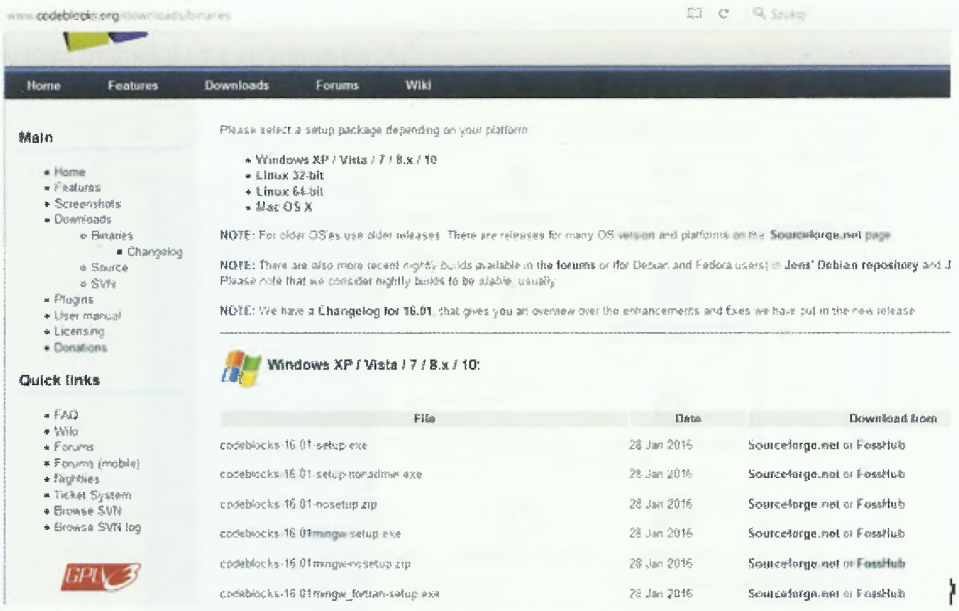
Lista dostępnych środowisk

- Dev-C++
  Darmowe środowisko programistyczne C/C++, które zawiera wielookienkowy edytor kodu źródłowego z podświetlaniem składni, kompilator, debbuger oraz linker. Od kilku lat nie jest ono już rozwijane. Edytor tego środowiska jest prosty w użyciu.
- Code::Blocks
  Darmowe środowisko programistyczne umożliwiające tworzenie aplikacji w języku C i C++. Program ten jest stale rozwijany. Dużą zaletą tego środowiska jest wieloplatformowość, tzn. środowisko jest dostępne zarówno w systemie operacyjnym Windows, jak i Linux. Zaraz po zainstalowaniu tego programu edytor jest skonfigurowany do pracy, więc od razu możemy przystępować do kodowania. Środowisko to możemy pobrać bez kompilatora lub z kompilatorem GCC od MinGW. Program zapewnia możliwość zaimportowania projektów utworzonych w innych środowiskach programowania.
- Microsoft Visual C++
  Płatne, rozbudowane środowisko programistyczne, które umożliwia tworzenie aplikacji dla systemu, ale jego złożoność może okazać się zbyt trudna dla początkujących programistów.

Podręcznik jest napisany z myślą o środowisku Code::Blocks, ponieważ:

- jest ono darmowe;
- jego możliwości są zbliżone do możliwości Visual C++;
- programu tego można używać zarówno w systemie Windows, jak Linuks;
- jest to dobre środowisko do edukacji.

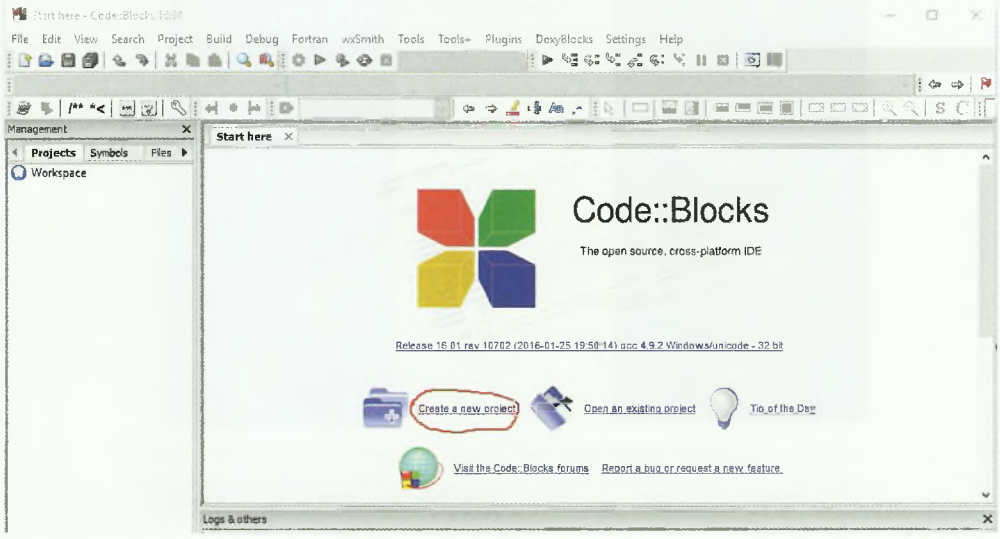
Instalacja Code::Blocks nie należy do skomplikowanych. Wystarczy pobrać ze strony http://www.codeblocks.org/ plik instalacyjny i go uruchomić. Należy pamiętać, aby pobierany plik instalacyjny zawierał pakiet MinGW (kompilator).



Rys. 2.1. Strona, zktórej można pobrać Code::Blocks

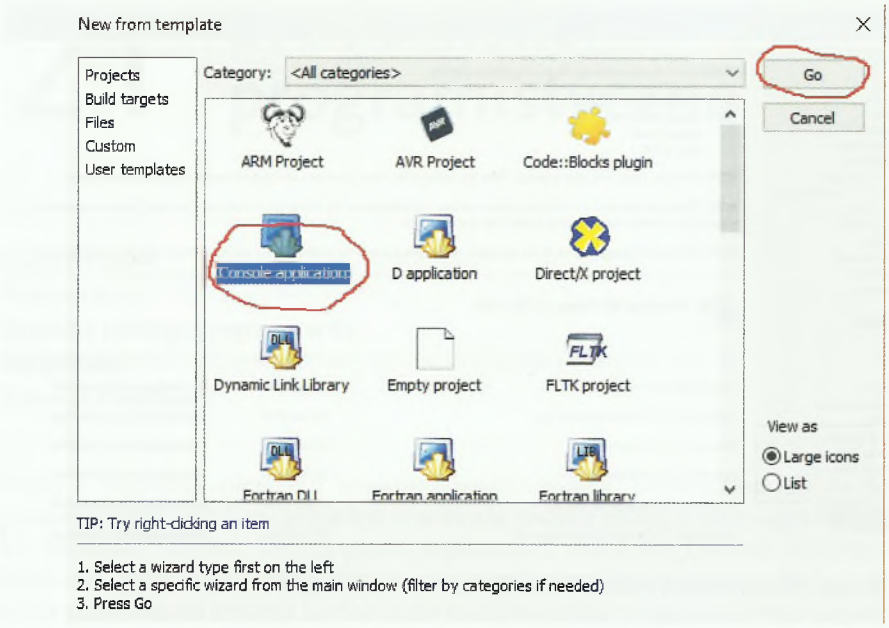
Kolejnym etapem instalacji jest pierwsze uruchomienie Code::Blocks. Na tym etapie należy wybrać kompilator, który będzie wykorzystywany do tłumaczenia aplikacji z języka C++ do postaci kodu wykonywalnego, tj. aplikacji w postaci pliku exe. Do paczki instalacyjnej Code::Blocks został dołączony kompilator **GNU GCC Compiler** i ten też powinien zostać wybrany.

Aby utworzyć nowy program w Code::Blocks, należy kliknąć: **FILE→NEW→PROJECT** lub bezpośrednio **Create a new project**.



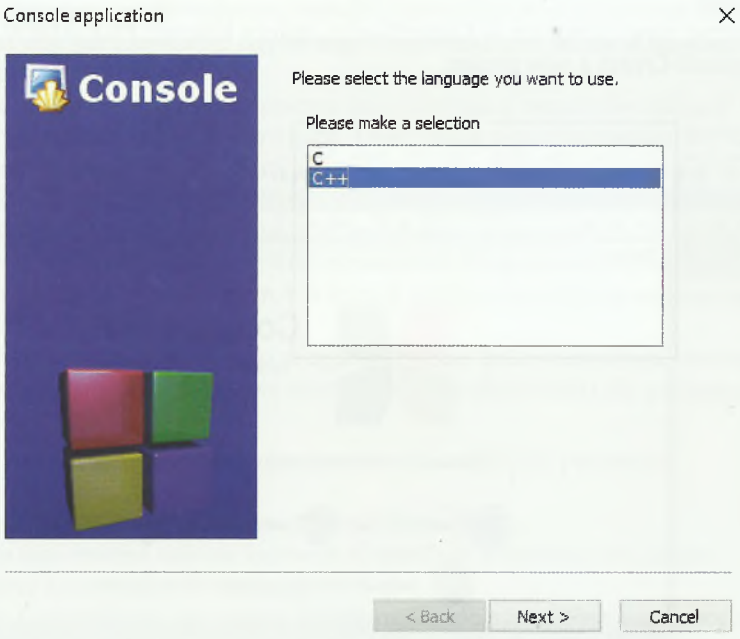
Rys. 2.2. Tworzenie nowego projektu w C++

Następnie trzeba kliknąć **Console application** i **Go**.



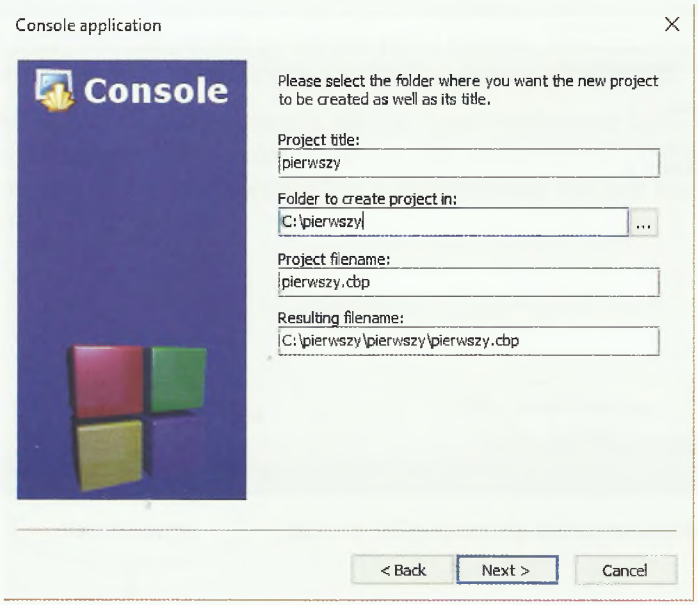
Rys. 2.3. Ciąg dalszy tworzenia nowego projektu w C++

W kolejnym kroku wybiera się język C lub C++ .



Rys. 2.4. Wybór języka programowania

W kolejnym oknie uzupełnia się nazwę projektu i określa się folder docelowy. Pozostałe wartości uzupełnią się automatycznie.

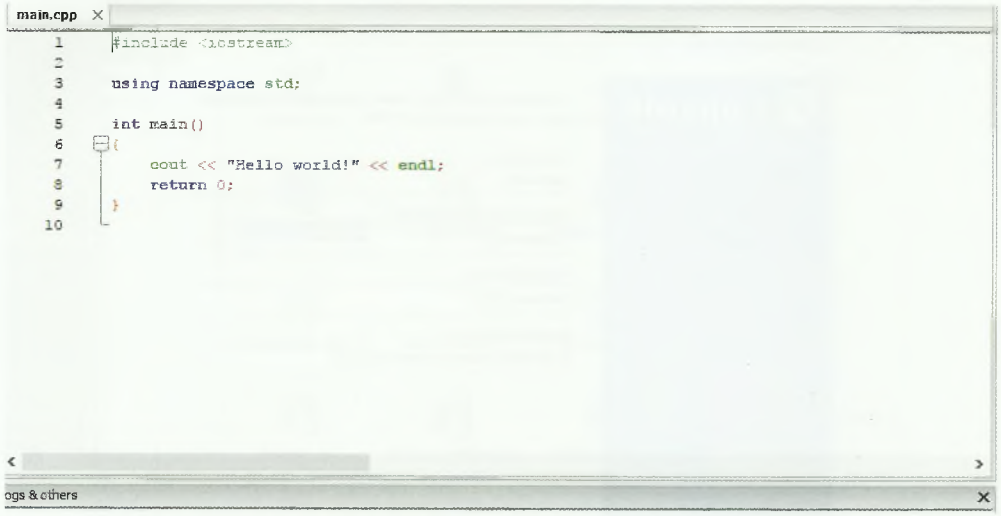


Rys. 2.5. Wybór nazwy projektu i jego folderu docelowego

Na koniec otrzymamy okno w następującej postaci:

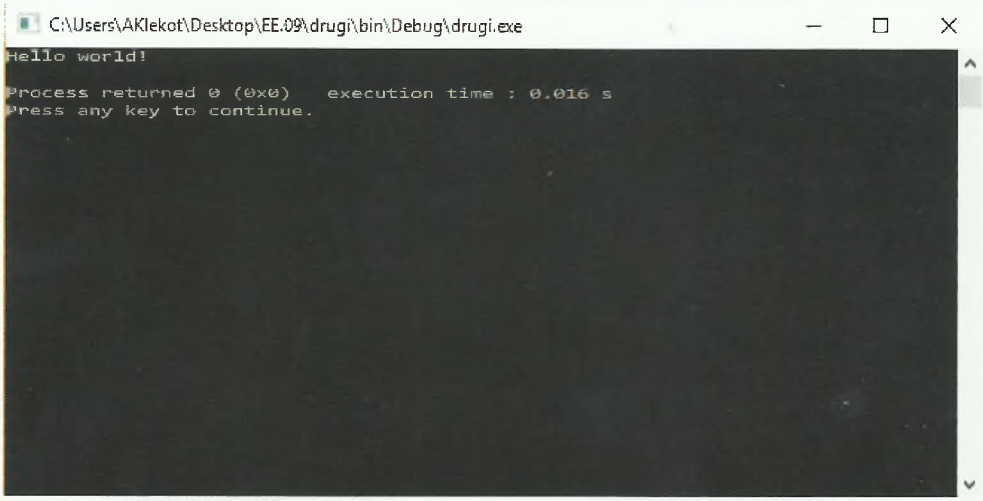
Rys. 2.6. Podsumowanie tworzenia nowego projektu

Naciśnięcie klawisza **Finish** spowoduje utworzenie nowego projektu z plikiem o nazwie **main.cpp**. Domyślna zawartość tego pliku jest następująca:



Rys. 2.7. Początkowa zawartość głównego pliku projektu

Aby uruchomić ten program, należy nacisnąć klawisz **F9**, którego zadaniem jest skompilowanie pliku ***.cpp** i przekształcenie go w tzw. plik wykonywalny. Otrzymamy okno w postaci:



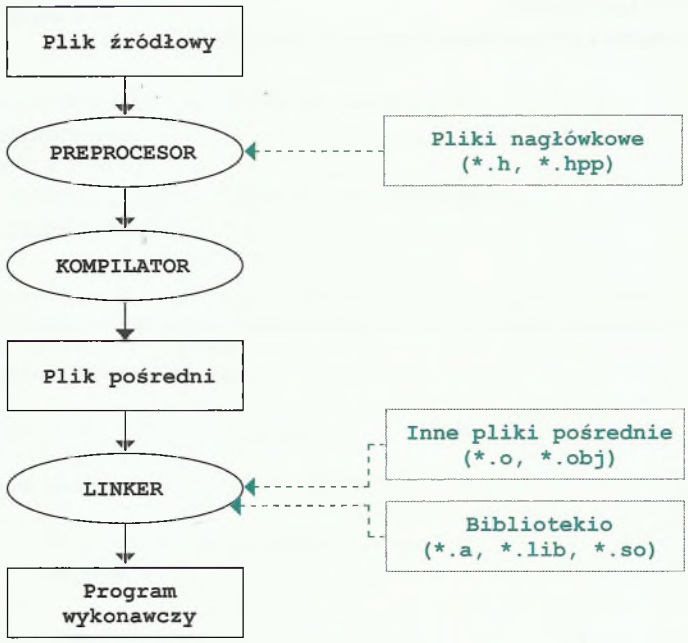
Rys. 2.8. Efekt działania programu po wciśnięciu F9

Kompilator to program tłumaczący kod źródłowy na plik wykonalny, a sam proces tłumaczenia nazywamy kompilacją.

Jest to tylko jeden z etapów procesu wytwarzania pliku wykonywalnego. W rzeczywistości są potrzebne jeszcze preprocesor i linker. Rolą linkera jest połączenie w jeden plik wykonywalny wielu plików otrzymanych w wyniku kompilacji poszczególnych plików źródłowych, których może być wiele i które mogą być kompilowane osobno i w różnym czasie.

Kompilacji oczywiście nie trzeba powtarzać przed każdym uruchomieniem programu. Raz utworzony plik wykonywalny można uruchamiać dowolną liczbę razy.

Wadą języków w pełni kompilowanych, jak C++, jest uzależnienie kodu wykonywalnego od platformy. Oczywiście sam tekst programu jest niezależny od platformy, jeśli trzymamy się w nim ściśle standardu języka C++ i nie stosujemy żadnych specyficznych dla danego kompilatora rozszerzeń.



Rys. 2.9. Schemat kompilacji

Jak pokazano na rys. 2.9, bez odpowiednich przekształceń kod źródłowy jest niezrozumiały dla komputera. Preprocesor jest technologią wykorzystywaną przez niektóre języki programowania, takie jak Perl, Python i oczywiście C++. Zadaniem jej jest translacja kodu źródłowego na **kod wyjściowy**. Jest to program komputerowy, który za pomocą tzw. dyrektyw preprocesora dokonuje istotnych (z punktu widzenia kompilatora) zmian. Preprocesor, korzystając z dyrektyw (w C++ są to #include, #define, ...), m. in. integruje kod z zawartością załączonych bibliotek standardowych, własnych plików nagłówkowych .h, tworzy makra i definiuje stałe.

W następnej kolejności kod ulega kompilacji, podczas której podlega analizom: leksykalnej, syntaktycznej oraz semantycznej. Kompilator sprawdza kod pod kątem błędów w pisowni słów kluczowych, obecności niedozwolonych znaków i błędów składniowych, a następnie sprawdza poprawność typów, użytych instrukcji i nazw.

Na etapie pracy konsolidatora (**linkera**) kod wynikowy łączony jest z innymi skompilowanymi modułami. Są nimi kody wynikowe dołączanych bibliotek oraz biblioteki uruchomieniowej, z której korzysta się w celu ograniczenia miejsca zajmowanego przez gotowy program. Efektem procesu konsolidacji jest gotowy do pracy w danym systemie plik wykonywalny.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Scharakteryzuj środowiska umożliwiające programowanie w C++.
2. Wymień cechy środowiska Code:Blocks.
3. Opisz proces instalacji Code:Blocks.
4. *Do czego służy kompilator?*
5. Omów rolę linkera i preprocesora w procesie kompilacji.

2.2 Struktura prostego programu w C++

ZAGADNIENIA

- **Części programu napisanego w C++**
- **Biblioteka w języku C++**

Program napisany w języku C++ składa się zazwyczaj z:

- dyrektyw preprocesora;
- definicji funkcji MAIN;
- deklaracji i definicji globalnych danych i struktur danych;
- deklaracji i definicji funkcji.

*main.cpp

```
1   #include <iostream>                    //obszar dyrektyw
2   using namespace std;
3
4   //obszar definicji i deklaracji zmiennych oraz funkcji
5
6   int main()                  //nagłówek funkcji main
7   {
8       cout << "Hello world!" << endl;
9       return 0;
10  }
```

Rys. 2.10. Struktura prostego programu

Jak pokazano na rysunku 2.10, na początku dołączamy potrzebne nam dyrektywy preprocesora. Są to wiersze zawierające specjalne instrukcje dla kompilatora lub przydatne funkcje. Każda dyrektywa rozpoczyna się znakiem #. Za koniec dyrektywy uważa się koniec wiersza, w związku z tym nie jest potrzebny średnik kończący polecenie. Każda dyrektywa musi występować w osobnej linii.

Najczęściej używaną dyrektywą jest dyrektywa **#include**. Jej działanie polega na dołączeniu do kompilowanego pliku zawartości innego pliku. Jest ona przydatna także w sytuacjach, kiedy chcemy skorzystać z funkcji bibliotecznych. Funkcje biblioteczne to najzwyklejsze funkcje dostarczone przez producenta kompilatora lub samodzielnie napisane przez programistę. Są one skompilowane i gotowe do pracy. Aby z nich skorzystać, należy dołączyć do programu określony plik nagłówkowy (tzw. bibliotekę), zawierający deklaracje interesujących nas funkcji.

Podstawowe biblioteki w języku C++:

- **math**
  Zawiera funkcje matematyczne, takie jak: funkcje trygonometryczne, logarytmy, zaokrąglanie liczb, pierwiastkowanie, potęgowanie.
- **stdio**
  Zawiera podstawowe operacje we/wy, czyli funkcje zarządzania plikami, obsługi klawiatury oraz ekranu.
- **iostream**
  Biblioteka obsługująca operacje wejścia i wyjścia.
- **stdlib**
  Zawiera funkcje zarządzania pamięcią, konwersję typów podstawowych, generowanie liczb losowych.
- **string**
  Zawiera funkcje zarządzania łańcuchami znaków.
- **time**
  Zawiera funkcje zarządzania czasem.

Składnia dyrektywy **#include** ma następującą postać:

#include <nazwa.h> lub **#include "plik.h"**

Załóżmy, że mamy dwa pliki źródłowe, w których chcemy użyć tej samej funkcji. Możemy zdefiniować tę funkcję dwa razy, w każdym z plików źródłowych. W takim przypadku jednak kompilator zaprotestuje, gdyż dana funkcja nie może być zdefiniowana dwa razy. Aby temu zapobiec, można funkcję zadeklarować w pliku nagłówkowym. Następnie plik ten dołączamy do obydwu plików źródłowych za pomocą **#include**.

Dyrektywa **#define** służy do zdefiniowania stałych. Definiowanie stałych odbywa się w następujący sposób:

```
#define nazwa_stałej wartość
```

Jeżeli preprocesor znajdzie w tekście programu nazwę jakiejś zdefiniowanej stałej, zamieni ją na jej wartość i **nie sprawdzi** przy tym, czy wygenerowany kod jest poprawny.

Jednak stała wcale nie musi mieć wartości. Może być po prostu zdefiniowana i nie mówić o sobie nic więcej.

Polecenie **using namespace std;** nakazuje użycia standardowej przestrzeni nazw **std**.

Pojęcie przestrzeni nazw służy do określenia, których zmiennych i funkcji można użyć w danym miejscu. Można tworzyć własne przestrzenie nazw, my jednak będziemy korzystać ze standardowej.

Następnie definiuje się funkcję o nazwie **main()**, wewnątrz której tworzymy kod programu. Na końcu tej funkcji można umieścić polecenie:

```
system(„pause");
```

Polecenie to zatrzymuje wykonanie programu do momentu naciśnięcia jakiegoś klawisza, co pozwala zobaczyć efekt działania programu.

Instrukcja **return 0;** powoduje, że funkcja **main()** zwraca wartość **0**, co oznacza, że program skompilował się pomyślnie.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Wymień części składowe programu napisanego w języku C++.
2. Omów funkcje biblioteki w języku C++.
3. Opisz składnię dyrektywy #include.

2.3 Typy zmiennych w C++

ZAGADNIENIA
- Co to jest zmienna?
- Typy zmiennych
- Klasy zmiennych

Zmienna to pewne miejsce w pamięci komputera, któremu można przypisać różne wartości. Każda zmienna ma swój adres w pamięci oraz nazwę. Przed użyciem zmiennej w programie należy ją zadeklarować. Deklaracja nie przypisuje wartości zmiennej, ale przydziela jej adres w pamięci (rezerwuje dla niej miejsce).

Przykładowa deklaracja może wyglądać następująco:

```
typ_zmiennej NazwaZmiennej;
```

Typy w języku C++ można podzielić na dwa sposoby:
- pierwszy podział dzieli typy na fundamentalne (podstawowe) i pochodne;
- drugi podział – na wbudowane i zdefiniowane przez użytkownika.

Podstawowe typy zmiennych to:
1) *char* – typ reprezentujący obiekty zadeklarowane jako znaki alfanumeryczne,
2) *int* – zmienna służąca do przechowywania liczb całkowitych,
3) *float* – zmienna służąca do przechowywania liczb rzeczywistych (zmiennoprzecinkowe, do 7 cyfr po przecinku),
4) *double* – zmienna służąca do przechowywania liczb rzeczywistych z dokładnością do 15 miejsc po przecinku.

W C++ można także użyć kwalifikatorów przed typem zmiennej:
a) *signed* (zmienna może przechowywać wartości dodatnie i ujemne – posiada znak +/-),
b) *unsigned* (zmienna może przechowywać tylko wartości dodatnie),
c) *short* (zmienna typu krótkiego),
d) *long* (zmienna typu długiego).

Rozmiar typów zmiennych przedstawia poniższa tabela:

Tabela 2.1. Rozmiary i zakresy typów danych

Typ zmiennej	Rozmiar (bajty)	Zakres wartości
char	1	od –128 do 127
int	2	od –32768 do 32767

Typ zmiennej	Rozmiar (bajty)	Zakres wartości
short	2	od –32768 do 32767
long	4	od –2147483648 do 2147483647
float	4	od $-3.4 \cdot 10^{38}$ do $3.4 \cdot 10^{38}$
double	8	od $-1.8 \cdot 10^{308}$ do $1.8 \cdot 10^{308}$

Istnieje także możliwość określenia klasy zmiennej. Typ zmiennej określa rozmiar oraz zakres wartości przechowywanych danych, natomiast klasa zmiennej decyduje o sposobie przechowywania zmiennej w pamięci. W C++ występują następujące klasy zmiennych:

- *const* – stała – nie można zmieniać wartości zmiennej,
- *static* – zmienna w momencie uruchomienia programu otrzymuje stałe miejsce w pamięci,
- *auto* – zmienna tego rodzaju musi zostać zainicjalizowana w momencie jej utworzenia. Po zakończeniu wykonywania instrukcji z danego bloku pamięć po zmiennej zostaje zwolniona.
- *register* – jest to informacja dla kompilatora o szybkim dostępie do zmiennej,
- *extern* – jeżeli zmienna została raz i tylko raz zadeklarowana w pojedynczym segmencie programu, zostanie w tym segmencie umieszczona w pamięci i potraktowana podobnie jak zmienna klasy *static*,
- *volatile* – (z ang. ulotny), zmienna może się zmienić w sposób niezauważalny dla kompilatora pomiędzy różnymi odczytami.

Nazwa zmiennej może być dowolnym ciągiem liter, cyfr i znaków podkreślenia. W nazwach zmiennych nie można używać polskich znaków i spacji oraz znaków specjalnych, a nazwa nie może zaczynać się od cyfry. Należy pamiętać, że kompilator rozróżnia małe i duże litery oraz że zmienne nie mogą mieć nazwy, która jest jednym ze słów kluczowych:

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, volitale, while.

Jeśli będziemy korzystać z kilku zmiennych tego samego typu, wystarczy podać tylko raz typ, a następnie – nazwy kilku zmiennych i oddzielić je przecinkami, np.

```
int zmienna1, zmienna2, zmienna3;
```

Wszystkie zmienne można podzielić na zmienne **globalne** i zmienne **lokalne**. Zmienne globalne są to takie zmienne, które są dostępne w **całym programie i przez cały czas jego działania**, natomiast zmienne lokalne są dostępne tylko w **pewnej części programu**, zazwyczaj tylko w **pewnej chwili działania programu**, a nie przez cały czas.

Zmienne globalne deklaruje się pomiędzy blokiem dołączonych plików nagłówkowych a funkcją **`main`**. Wszystkie zmienne globalne są widoczne wewnątrz funkcji **`main`**. Oznacza to, że wewnątrz funkcji **`main`** (oraz wszystkich innych funkcji) można dokonywać wszystkich operacji, jakie są tylko możliwe dla zmiennej danego typu. Trzeba jednak pa-

miętać, że przy tworzeniu tego typu zmiennych istnieje niebezpieczeństwo przypadkowego nadpisania ich wartości, co może spowodować nieprawidłowe działanie programu. Dlatego zaleca się korzystanie ze zmiennych lokalnych, które można deklarować wewnątrz funkcji lub w danym bloku programu.

PRZYKŁAD 2.1

```
#include <iostream>

using namespace std;
int liczba1;          //zmienna globalna
float liczba2;        //zmienna globalna
int main()
{
    float wiek;         //zmienna lokalna
    cout << "Ile masz lat?" << endl;
    cin>>wiek;
    return 0;
}
```

SPRAWDŹ SWOJĄ WIEDZĘ

1. Wymień typy zmiennych w C++.
2. Scharakteryzuj klasy zmiennych.

2.4 Operatory w C++

ZAGADNIENIA

- **Co to jest operator w C++?**
- **Rodzaje operatorów C++**

Operator to znak lub ciąg znaków (zazwyczaj niebędących literami), które mają specjalne znaczenie w języku programowania.

W C++ wyróżnia się kilka rodzajów operatorów.

1. **Operatory arytmetyczne** – wykonują działania na dwóch obiektach, czyli są to operatory dwuargumentowe:
   a) + operator dodawania;
   b) – operator odejmowania;
   c) * operator mnożenia;
   d) / operator dzielenia;
   e) % operator modulo (wyznacz resztę z dzielenia).
2. **Operatory porównania (relacji)** – są także operatorami dwuargumentowymi:
   a) > większy od;
   b) < mniejszy od;
   c) >= większy od lub równy;
   d) <= mniejszy od lub równy;
   e) == równy;
   f) != różny od.
3. **Operatory logiczne:**
   a) ! negacja;
   b) && iloczyn logiczny (koniunkcja);
   c) || suma logiczna (alternatywa).
4. **Operatory przypisania:**
   a) = zwykłe przypisanie;
   b) += dodaj i przypisz;
   c) –= odejmij i przypisz;
   d) *= pomnóż i przypisz;
   e) /= podziel i przypisz.
5. **Operatory inkrementacji i dekrementacji** – czyli operatory zwiększania wartości zmiennej o jeden i zmniejszania wartości o jeden. Są to operatory jednoargumentowe.
   Działanie typu `i=i+1` można zastąpić zapisem `i++`, natomiast działanie `i=i-1` można zastąpić zapisem `i--`.
   Jeśli zatem zmiennej `i` przypiszemy wartość `5`, to instrukcja `i++` zmieni wartość zmiennej na liczbę `6`.

6. **Operatory bitowe:**
   a) << przesunięcie w lewo;
   b) >> przesunięcie w prawo;
   c) & bitowy iloczyn logiczny;
   d) | bitowa suma logiczna;
   e) ^ bitowa różnica symetryczna;
   f) ~ bitowa negacja.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Wymień i omów operatory w C++.
2. Do czego służy operator modulo?
3. Wymień i omów operatory jednoargumentowe.

2.5 Instrukcje wejścia/wyjścia

ZAGADNIENIA
- **Funkcje instrukcji we / wy**
- **Komentarze**

Do wyświetlania danych na ekranie służy funkcja **cout**, zdefiniowana w bibliotece **iostream** (out jak wyjście, np. na ekran monitora). Jego konstrukcja jest następująca:

```
std::cout << „Treść komunikatu";
```

Jeśli do programu dołączymy linijkę: **using namespace std**; zapis możemy skrócić do postaci: **cout << „Treść komunikatu";**

Powyższa konstrukcja pozwala wyświetlać na ekranie monitora napis, który jest umieszczony w cudzysłowie. Dodatkowym elementem jest operator wyjścia <<, który pokazuje kierunek przekierowania strumienia znaków.

Aby wyświetlić wartość zmiennych lub działań, zapisujemy je bez użycia cudzysłowu, np.:

```
cout << 2*b + 2*a;
```

Możemy także łączyć treść komunikatu z wartościami zmiennych lub wyrażeniami. Każde z nich oddzielamy operatorem <<, np.:

```
cout <<"Liczba "<<3*7<<" jest większa niż "<<2*7;
```

Na ekranie monitora otrzymamy wówczas następującą informację:

```
Liczba 21 jest większa niż 14
```

Manipulator strumienia **endl** wstawia znak nowej linii i przenosi do niej kursor.

Do pobierania danych z klawiatury służy funkcja **cin**, która jest również zdefiniowana w bibliotece **iostream**.

Tak jak w przypadku obiektu **cout**, mamy do dyspozycji dwie możliwości:

```
std::cin>>nazwa_zmiennej;
```

lub po dodaniu linijki

```
using namespace std;
```

mamy postać:

```
cin>>nazwa_zmiennej;
```

W tym przypadku stawiamy znaki przekierowania strumienia w odwrotną stronę >>.

Jeśli chcemy podać wartości kilku zmiennych, należy oddzielić zmienne znakiem wejścia strumienia, np.:

```
cin>> zmienna1>> zmienna2>> zmienna3;
```

Do wczytywania i wypisywania danych na ekranie komputera służą także instrukcje **printf** oraz **scanf**. Działają one znacznie szybciej niż **cout** i **cin** oraz dostarczają więcej możliwości programiście.

Aby móc skorzystać z operacji wejścia/wyjścia **printf** oraz **scanf**, należy dodać bibliotekę **stdio**.

Pierwsza z operacji ma następującą postać:

```
printf(„Treść komunikatu");
```

Jeśli w środek tekstu ujętego w cudzysłów wpiszemy znaki **\n**, to efekt programu będzie następujący:

```
Treść
komunikatu
```

Uwaga!

Znak **\n** (n – jak: new line, czyli: nowa linia) powoduje, że w trakcie wypisywania tekstu na ekranie następuje przejście do nowej linii i dalszy ciąg tekstu jest wypisywany poniżej. Oprócz niego można zastosować następujące znaki:

- **\t** wstawienie znaku tabulacji;
- **\'** wstawienie apostrofu;
- **\"** wstawienie cudzysłowu;
- **\?** wstawienie znaku zapytania.

Każdy porządny programista stosuje podczas pisania kodu źródłowego tzw. komentarze. Są to teksty ignorowane przez komputer i służą do opisu danej części kodu oraz dokumentacji technicznej programu. W C++ komentarze można zapisać w dwojaki sposób:

- jako komentarz podwójnego ukośnika **//**
- jako ukośnik i gwiazdka **/***

Komentarze podwójnego ukośnika ignorują tekst do końca linii, natomiast komentarze ukośnika i gwiazdki ignorują tekst aż do znaku gwiazdki i ukośnika ***/**.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Przytocz formuły instrukcji **cout** oraz **cin**.
2. Na czym polegają różnice między **printf** a **cout**?
3. W jaki sposób można tworzyć komentarze w C++?

3. Instrukcje sterujące

- Instrukcja warunkowa
- Instrukcja SWITCH
- Pętla WHILE
- Pętla DO WHILE
- Pętla FOR

3.1 Instrukcja warunkowa

ZAGADNIENIA
- Niepełna instrukcja warunkowa
- Pełna instrukcja warunkowa

Instrukcje sterujące służą do sterowania przebiegiem programu. Dzięki nim są podejmowane decyzje o wykonaniu tych czy innych instrukcji w programie. Decyzje te zależą od wyniku sprawdzenia określonych warunków, czyli od ustalenia, czy warunek jest prawdziwy (przyjmuje wartość różną od zera – PRAWDA) lub fałszywy (przyjmuje wartość 0 – FAŁSZ).

W języku C++ instrukcje zawsze kończymy średnikiem (instrukcja może być zapisana w kilku wierszach). Najprostsza instrukcja składa się tylko ze średnika. Nazywa się ją instrukcją pustą. Jest ona użyteczna na przykład w niektórych pętlach.

Najczęściej używaną instrukcją jednak jest tzw. instrukcja przypisania, która ma następująca postać:

```
zmienna = działanie;
```

PRZYKŁAD 3.1

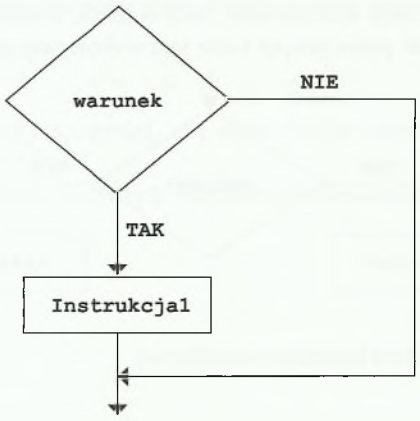
```
x=a+b;
```

Wszystko to, co staje się wartością, w C++ jest uważane za wyrażenie. Każde wyrażenie zwraca wartość. Powyższy przykład oznacza, że do zmiennej `x` zostanie przypisana wartość sumy zmiennych `a` i `b`.

Instrukcja warunkowa może mieć dwie postaci: niepełną i pełną. Instrukcja warunkowa umożliwia kontrolę wykonywania poszczególnych instrukcji w obrębie programu w zależności od spełnienia warunku.

Warunek jest to wyrażenie, które ma jakąś wartość. Jeśli jest ona niezerowa, czyli ma wartość typu PRAWDA, to jest wykonywana Instrukcja1. W przeciwnym wypadku Instrukcja1 nie jest wykonywana, następuje zakończenie programu lub przejście do kolejnego bloku instrukcji. W ten sposób działa niepełna instrukcja warunkowa.

Niepełna instrukcja warunkowa



Rys. 3.1. Schemat blokowy niepełnej instrukcji warunkowej

Postać instrukcji:

```
if (warunek)
{
Instrukcja1;
}
```

PRZYKŁAD 3.2

Napisz program obliczający pole kwadratu.
Musimy pamiętać, że długości boków każdej figury geometrycznej są liczbami dodatnimi. Należy więc sprawdzić, czy podana przez użytkownika liczba ma wartość większą od zera. Jeśli TAK, to zostanie obliczone pole kwadratu, w przeciwnym razie nastąpi zakończenie programu.

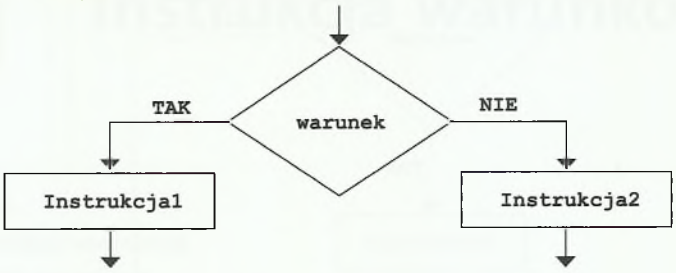
```
#include <iostream>
using namespace std;
int main()
{
float a,p;
cout<<"Podaj długość boku kwadratu: ";
cin>>a;
if (a>0)
    {
        p=a*a;
        cout<<"Pole kwadratu wynosi: "<<p;
    }
return 0;
}
```

Rys. 3.2. Zastosowanie niepełnej instrukcji warunkowej

Uwaga!
Przy standardowej konfiguracji programu Code:: Block polskie znaki mogą nie być wyświetlane poprawnie.

Pełna instrukcja warunkowa

W przypadku pełnej instrukcji warunkowej, jeśli wartość Warunku jest niezerowa, to jest wykonywana Instrukcja1. W przeciwnym razie jest wykonywana Instrukcja2.



Rys. 3.3. Schemat blokowy pełnej instrukcji warunkowej

Postać instrukcji:

```
if (warunek)
{
Instrukcja1;
}
else
{
Instrukcja2;
}
```

PRZYKŁAD 3.3

Napisz program obliczający pole kwadratu. Użycie pełnej instrukcji warunkowej w tym przykładzie poprawi działanie programu, ponieważ w przypadku gdy użytkownik poda ujemną wartość boku kwadratu, zostanie wyświetlony na ekranie komunikat o źle wprowadzonej wartości.

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

int main()
{
    float a,p;
    cout<<"Podaj dlugosc boku kwadratu: "<<endl;
    cin>>a;
    if (a>0)
    {
        p=a*a;
        cout<<"Pole kwadratu wynosi: "<<p;
    }
       else
    {
        cout<<"Dlugosc boku kwadratu musi byc dodatnia.";
    }
    return 0;
}
```

Rys. 3.4. Zastosowanie pełnej instrukcji warunkowej

Konstrukcja **warunków instrukcji if** może być prosta – składająca się z jednego warunku – lub złożona z kilku warunków połączonych operatorami logicznymi.

PRZYKŁAD 3.4

Napisz program, który sprawdzi, czy dana liczba naturalna jest liczbą parzystą i niepodzielną przez 3.

Można zastosować dwie instrukcje warunkowe zawierające warunki proste:

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

int main()
{
  int liczba;

  cout<<"Podaj liczbę: ";
  cin>>liczba;

  if(liczba%2==0)       //warunek prosty
    {
        if (liczba%3!=0)     //warunek prosty
            {
                cout<<"Liczba "<<liczba<<" jest liczba parzysta niepodzielna przez 3."<<endl;
            }
        else
            {
                cout<<"Liczba "<<liczba<<" nie jest jednoczesnie liczba parzysta i niepodzielna przez 3."<<endl;
            }
    }
    else
    {
        cout<<"Liczba "<<liczba<<" nie jest jednoczesnie liczba parzysta i niepodzielna przez 3."<<endl;
    }
  return 0;
}
```

Rys. 3.5. Realizacja zadania z wykorzystaniem warunków prostych

Ten sam program można zrealizować, stosując warunek złożony:

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

int main()
{
  int liczba;

  cout<<"Podaj liczbę: ";
  cin>>liczba;

  if(liczba%2==0 && liczba%3!=0)    // warunek złożony z dwóch warunków połączonych operatorem logicznym &&
      cout<<"Liczba "<<liczba<<" jest liczba parzysta niepodzielna przez 3."<<endl;
    else
      cout<<"Liczba "<<liczba<<" nie jest jednoczesnie liczba parzysta i niepodzielna przez 3."<<endl;

  return 0;
}
```

Rys. 3.6. Realizacja zadania z wykorzystaniem warunku złożonego

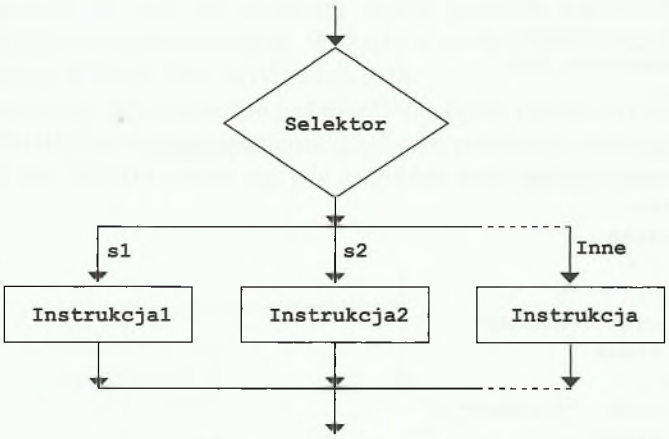
SPRAWDŹ SWOJE UMIEJĘTNOŚCI

1. Napisz program, który wykonuje mnożenie dwóch podanych przez ciebie liczb, przypisuje wynik pewnej zmiennej, a następnie sprawdza, czy dana liczba jest większa od 100, równa 100, czy mniejsza od 100.
2. Napisz program, który wykona dodawanie dwóch wpisanych z klawiatury liczb, ale tylko w przypadku, gdy NIE będą sobie równe. W przeciwnym razie wykona operację dzielenia pierwszej przez drugą.
3. Napisz program, który pobiera wartości dla 3 zmiennych i wykonuje:
   - mnożenie liczby pierwszej oraz drugiej, gdy liczba pierwsza jest większa od trzeciej i liczba druga jest większa od pierwszej;
   - dzielenie drugiej liczby przez trzecią, gdy liczba druga jest mniejsza od trzeciej albo mniejsza od pierwszej;
   - dodawanie wszystkich trzech liczb w przypadku, gdy liczba trzecia jest większa od pierwszej i liczba druga nie jest równa 5 lub liczba druga jest większa od trzeciej oraz liczba pierwsza nie jest równa 0.

3.2 Instrukcja SWITCH

ZAGADNIENIA
- Funkcja instrukcji SWITCH
- Schemat blokowy instrukcji SWITCH

Instrukcja SWITCH służy do podejmowania wielowariantowych decyzji.



Rys. 3.7. Schemat blokowy instrukcji SWITCH

Selektor jest to zmienna, która może przyjmować różne wartości. Jeśli wartość Selektora odpowiada wartości podanej w jednej z etykiet **case** (s1, s2, ...), wtedy instrukcje są wykonywane począwszy od tej etykiety. Wykonywanie kończy się po napotkaniu instrukcji **break** i następuje wyjście z instrukcji **switch**.

Jeśli wartość Selektora nie zgadza się z żadną z wartości podanych przy etykietach **case**, to wykonują się instrukcje umieszczone po etykiecie **default** (może znajdować się ona w dowolnym miejscu instrukcji **switch**).

Instrukcja **break** powoduje natychmiastowe przerwanie wykonywania instrukcji. Jeśli mamy do czynienia z kilkoma pętlami – zagnieżdżonymi jedna wewnątrz drugiej, to instrukcja **break** powoduje przerwanie tylko tej pętli, w której bezpośrednio tkwi.

Postać instrukcji:

```
switch (Selektor)
{
case s1:
     Instrukcja1;
     break;
```

```
case s2:
     Instrukcja2;
     break;
.........
default:
     Instrukcja;
     break;}
```

PRZYKŁAD 3.5

Napisz program, który:

- wypisze na ekranie komunikat „Monday", jeśli użytkownik wpisze liczbę 1,
- wypisze na ekranie komunikat „Tuesday", jeśli użytkownik wpisze liczbę 2,
- wypisze na ekranie komunikat „Wednesday", jeśli użytkownik wpisze liczbę 3,
- wypisze na ekranie komunikat „Podałeś złą liczbę", jeśli użytkownik wpisze każdą inną liczbę.

```
#include <iostream>
using namespace std;
int main()
{
int dzien;                              //Selektor
cout<<"Podaj cyfre od 1 do 3";
cin>>dzien;
switch(dzien)
    {
    case 1:
        cout<<"Monday";
        break;
    case 2:
        cout<<"Tuesday";
        break;
    case 3:
        cout<<"Wednesday";
        break;
    default:
        cout<<"Zla liczba";
        break;
    }
return 0;
}
```

Rys. 3.8. Zastosowanie instrukcji SWITCH

SPRAWDŹ SWOJĄ WIEDZĘ

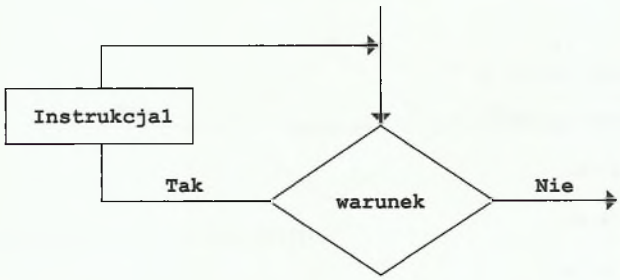
1. Do czego służy instrukcja SWITCH?
2. Opisz schemat blokowy instrukcji SWITCH.
3. Czym różni się instrukcja IF od instrukcji SWITCH?

3.3 Pętla WHILE

ZAGADNIENIA
- Funkcja pętli WHILE
- Schemat blokowy instrukcji WHILE

Pętla to konstrukcja programowania, która umożliwia cykliczne wykonywanie ciągu instrukcji określoną liczbę razy, do momentu zajścia pewnych warunków, dla każdego elementu kolekcji lub w nieskończoność. W języku C++ do zbudowania pętli można wykorzystać instrukcje WHILE, DO...WHILE lub FOR.

Pętli FOR używa się, gdy konieczne jest określenie liczby powtórzeń danego bloku instrukcji. Pętlę WHILE natomiast stosujemy, gdy liczba powtórzeń zależy od warunku pętli. Z kolei pętli DO...WHILE używa się, gdy dany blok musi być wykonany przynajmniej jeden raz.



Rys. 3.9. Schemat blokowy instrukcji WHILE

Tak jak w przypadku instrukcji warunkowej, najpierw oblicza się wartość Warunku. Jeśli wynik jest zerowy, to następuje wyjście z pętli. Jeśli wynik jest wartością niezerową, jest wykonywana Instrukcja1, a następnie ponownie jest sprawdzana wartość warunku. W instrukcji tej nie określamy liczby powtórzeń Instrukcji1. Wykonanie bloku instrukcji wewnątrz pętli odbywa się tylko wtedy, gdy wartość warunku jest niezerowa.

Postać instrukcji:

```
while (Warunek)
{
Instrukcja1;
}
```

PRZYKŁAD 3.6

Napisz program, który wypisze liczby od 1 do 25 w postaci kolumnowej.

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

int main()
{
 int i=1;
 while (i<=25)
 {
      cout<<i<<endl;
      i++;

 }
return 0;
}
```

Rys. 3.10. Zastosowanie pętli WHILE

PRZYKŁAD 3.7

Klasycznym zastosowaniem pętli WHILE jest wyznaczenie największego wspólnego dzielnika (NWD), np. dla dwóch liczb podanych z klawiatury:

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

int main()
{
  int a,b;
  cout<<"Podaj liczbe a: ";
  cin>>a;
  cout<<"Podaj liczbe b: ";
  cin>>b;
    while (a!=b)
    {
        if (a<b)
        {
            b-=a;
        }
        else
        {
            a-=b;
        }
    }
  cout<<"Najwiekszy wspolny dzielnik wynosi: "<<a<<endl;
  return 0;
}
```

Rys. 3.11. Wyznaczenie NWD dla dwóch liczb całkowitych

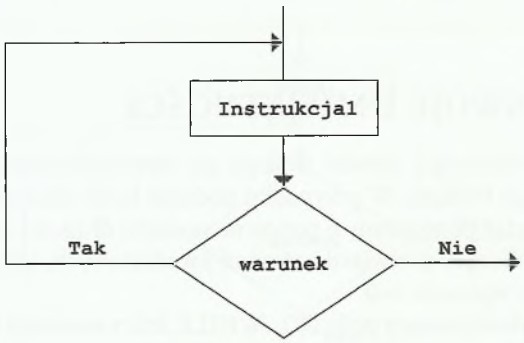
SPRAWDŹ SWOJĄ WIEDZĘ

1. Scharakteryzuj schemat blokowy instrukcji WHILE.
2. Podaj składnię pętli WHILE.
2. Kiedy konstrukcja wewnątrz pętli WHILE nie zostanie wykonana ani razu? Podaj konkretny przykład.

3.4 Pętla DO...WHILE

ZAGADNIENIA
- **Funkcja pętli DO...WHILE**
- **Schemat blokowy instrukcji DO...WHILE**

Słowa DO...WHILE oznaczają w języku angielskim 'rób... dopóki'. W pętli DO...WHILE jest wykonywana **Instrukcja1**, a dopiero potem następuje sprawdzenie warunku. Jeśli warunek zostaje spełniony, to **Instrukcja1** nie jest już więcej powtarzana. Jeśli warunek nie został spełniony, to **Instrukcja1** nie zostanie już wykonana ani razu.



Rys. 3.12. Schemat blokowy instrukcji DO...WHILE

Konstrukcja **pętli DO...WHILE** pozwala na wykonanie przynajmniej raz instrukcji wewnątrz pętli pomimo niespełnionego warunku pętli. Natomiast instrukcja wewnątrz pętli WHILE nie wykonana się ani razu, gdy warunek pętli jest zerowy.

Postać pętli:

```
do
{
Instrukcja1;
}
while (warunek);
```

PRZYKŁAD 3.8

Napisz program, który wypisze ciąg malejących liczb całkowitych od 25 do 1 w postaci kolumnowej:

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6   int i=25;
7   do
8   {
9       cout<<i<<endl;
10      i--;
11  }while (i>=1);
12  return 0;
13  }
14
```

Rys. 3.13. Zastosowanie pętli DO...WHILE

SPRAWDŹ SWOJE UMIEJĘTNOŚCI

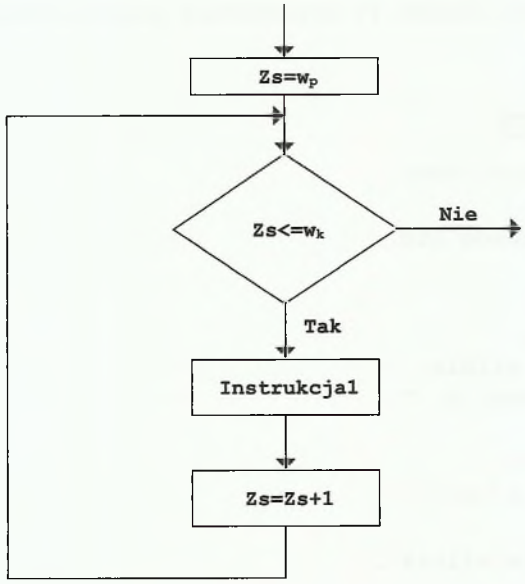
1. Napisz program obliczający obwód trójkąta po wprowadzeniu przez użytkowników długości boków tego trójkąta. W przypadku podania liczb ujemnych lub równych zero użytkownik ma zostać poproszony o ponowne podanie długości boków.
2. Napisz program obliczący pierwiastki równań kwadratowych, tak by pozwalał na kolejne obliczenia aż do wpisania a=0.
3. Napisz program wykorzystujący pętlę DO...WHILE, który wyświetli liczby całkowite od 25 do 100 i od 25 do 200.

3.5 Pętla FOR

ZAGADNIENIA

- **Funkcja pętli FOR**
- **Schemat blokowy instrukcji FOR**

Do wykonywania pętli o określonej liczbie powtórzeń stosuje się instrukcję FOR. Ma ona trzy argumenty. Pierwszy z nich **Zs=w_p** określa wartość początkową zmiennej sterującej pętli (**Zs** – zmienna sterująca). Warunek **Zs<w_k** wyznacza wartość końcową zmiennej sterującej, natomiast instrukcja **Zs++** – sposób zmiany wartości zmiennej sterującej (w tym przypadku inkrementacja o 1).



Rys. 3.14. Schemat blokowy instrukcji FOR

Instrukcja FOR ma następującą postać:

```
for (Zs=wp; Zs<wk; Zs++)
{
Instrukcja1;
}
```

PRZYKŁAD 3.9

Napisz program, który 5 razy wypisze na ekranie komunikat „Programowanie w C++".

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

int main()
{
    for (int i=1;i<=5;i++)
    {
        cout<<"Programowanie w C++"<<endl;
    }
    return 0;
}
```

Rys. 3.15. Zastosowanie pętli **FOR**

W rozdziale 1 przedstawiono algorytm iteracyjny obliczający silnię podanej przez użytkownika liczby (rys. 1.4). Pętla FOR jest stworzona do operacji iteracyjnych z określoną liczbą wykonywanych obliczeń. Po implementacji program powinien wyglądać tak jak w przykładzie 3.10 .

PRZYKŁAD 3.10

```
#include <iostream>

using namespace std;

int main()
{
  int i, n,silnia;
  cout<<"Podaj n: ";
  cin>>n;
    silnia=1;
  for (i=1;i<=n;i++)
  {
      silnia=silnia*i;
  }
  cout<<"Silnia wynosi: "<<silnia;
  return 0;
}
```

Rys. 3.16. Obliczanie silni za pomocą pętli FOR

SPRAWDŹ SWOJE UMIEJĘTNOŚCI

1. Napisz program, który wypisze w formie kolumny liczby całkowite od 1 do 25 i odwrotnie.
2. Napisz program, który obliczy sumę 10 dowolnych liczb podanych przez użytkownika.
3. Napisz program, który spośród 8 liczb podanych przez użytkownika wyszuka te, które są podzielne przez 3, i obliczy ich sumę.
4. Napisz program, który spośród liczb naturalnych mniejszych od 100 wyszuka i wypisze liczby podzielne przez 5 i jednocześnie niepodzielne przez 3.
5. Napisz program, który wypisze parzyste liczby naturalne mniejsze od 100. Zadanie wykonaj, wykorzystując każdą z trzech poznanych pętli.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Omów postać pętli FOR.
2. Przedstaw schemat blokowy pętli FOR.

4. Funkcje

- Budowa i rodzje funkcji
- Wywoływanie funkcji i przekazywanie argumentów
- Przeładowanie funkcji
- Argumenty domniemane

4.1 Budowa funkcji

ZAGADNIENIA

- Co to jest funkcja?
- Definicja funkcji
- Prototyp funkcji
- Funkcja, która nie zwraca wartości i nie ma argumentów
- Funkcja mająca argumenty

Jedną z najciekawszych cech współczesnych języków programowania jest możliwość posługiwania się podprogramami.

Pewne instrukcje są wykonywane w danym programie wielokrotnie dla tych samych lub innych zmiennych. Dlatego też dobrze jest zgrupować powtarzające się obliczenia pod jedną nazwą. Tworzymy w ten sposób **funkcję**, czyli podprogram wykonujący pewne zadanie na potrzeby programu głównego. Jest to fragment programu, któremu nadano nazwę i który możemy wykonać przez podanie jego nazwy oraz ewentualnych argumentów (o ile istnieją).

Każda funkcja musi mieć:

- nazwę;
- typ zwracanej przez nią wartości;
- dowolną liczbę argumentów.

Funkcja może pobierać argumenty z programu głównego i użyć ich do wykonania określonych zadań.

Jeżeli funkcja zwraca jakąś wartość, należy określić typ zwracanej wartości. Sposób wywoływania funkcji zależy od jej typu. Zanim jednak ją wywołamy, musimy wiedzieć, czy pobiera argumenty z programu głównego i czy zwraca jakąś wartość. Każda funkcja, która zwraca wartość, musi być wywoływana przy jednoczesnym przypisaniu danej wartości pewnej zmiennej globalnej.

Funkcja, która nie zwraca wartości i nie ma argumentów

Weźmy pod uwagę następującą funkcję:

```
void wypisz()
{
    //cialo funkcji

}
```

Nazwą funkcji jest **wypisz**, a funkcja nie przyjmuje żadnych argumentów, ponieważ wewnątrz zaokrąglonych nawiasów jest pusto. Między klamrami umieszczamy kod, który ma zostać wykonany, gdy zostanie wywołana funkcja. Kod ten nazywa się **ciałem funkcji**.

Słowo kluczowe **void** przed nazwą funkcji informuje nas, że funkcja nie zwraca żadnych danych, które można byłoby przekazać innej funkcji czy instrukcji. Funkcja tego typu może wykonywać pewne czynności, ale nie przekazuje informacji zwrotnej. Funkcja **wypisz** może na przykład wyświetlać komunikat: „Witaj!". Oprócz wyświetlanej informacji żadne dane nie będą dalej przekazywane.

Gdyby jednak funkcja **wypisz** miała za zadanie na przykład wypisać resztę z dzielenia podanej liczby przez 2, to typ **void** nie będzie poprawny. Informacją zwrotną będzie liczba, która jest resztą z dzielenia, czyli wartość typu **int**.

Funkcja mająca argumenty

Wartością zwrotną funkcji jest wyliczona wartość, która jest przekazywana do dalszych działań. Jej typ wypisujemy przed definicją naszej funkcji i jest ona zwracana za pomocą słowa kluczowego **return**.

```
typ_zwracanej_wartosci nazwa_funkcji(typ_arg_1 nazwa_arg_1/*,..*/ typ_arg_n nazwa_arg_n)
{
    return zwracana_wartosc;
}
```

Argumenty określają, jakiego typu zmienne należy przekazać do funkcji przy jej wywoływaniu.

Są to dane, na podstawie których zostaną wykonane instrukcje wewnątrz funkcji.

PRZYKŁAD 4.1

```
void sprawdz(int a)
{
    if (a%2==0)
    {
        cout<<"Liczba "<<a<<" jest parzysta";
    }
    else
    {
        cout<<"Liczba "<<a<<" jest nieparzysta";
    }
}
```

Rys. 4.1. Przykład funkcji jednoargumentowej nie zwracającej wartości

Została zadeklarowana funkcja **sprawdz**, która ma jeden argument typu całkowitego oraz nie zwraca żadnej wartości, czyli zwraca typ **void**. Funkcja **sprawdz** sprawdza, czy podana liczba jest parzysta, czy nie i wyświetla odpowiedni komunikat.

Funkcja o nazwie **wyrażenie** mająca trzy argumenty typu **float** i zwracająca wartość typu **float**.

PRZYKŁAD 4.2

```
float wyrazenie(float x, float y, float z)
{
    float wyr=x+2*y-sqrt(z);
    return wyr;
}
```

Rys. 4.2. Przykład funkcji z trzema argumentami i zwracającej wartość

Należy pamiętać, że funkcja, która zwraca wartość, musi zwracać ją zawsze. Jeżeli funkcja dojdzie do końca bloku funkcji i nie napotka słowa kluczowego **return**, zachowanie wyjścia z funkcji jest niezdefiniowane. Oznacza to, że aplikacja może zakończyć pracę w wyniku wystąpienia błędu krytycznego aplikacji.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Wymień własności funkcji.
2. O czym informuje słowo kluczowe **void**?
3. Jaką funkcję pełni słowo kluczowe **return**?

4.2 Wywoływanie funkcji i przekazywanie argumentów do funkcji

ZAGADNIENIA

- Sposób wywoływania funkcji
- Funkcja główna `main`
- Funkcja `suma`

Aby wywołać w programie przygotowaną wcześniej funkcję, należy po prostu napisać jej nazwę w odpowiednim miejscu, a w nawiasach okrągłych – argumenty tej funkcji (jeśli je ma). Wywoływana funkcja musi zawsze być zadeklarowana przed miejscem, gdzie została wywołana. Można przed wywołaniem funkcji napisać tylko jej deklarację, a definicję (ciało) – w dowolnym miejscu programu.

PRZYKŁAD 4.3

Rozpatrzmy program, który będzie wykonywał dodawanie dwóch liczb rzeczywistych. Działanie to będzie wykonywała funkcja **suma**.

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

float suma(float a,float b)          //definicja funkcji suma zwracającej wynik typu float
{                                    //funkcja posiada dwa argumenty typu float
    return a+b;
}
int main()
{
    float x,y;
    cout<<"Podaj pierwszą liczbę: ";
    cin>>x;
    cout<<"Podaj drugą liczbę: ";
    cin>>y;
    cout<<"Suma liczby "<<x<<" i "<<y<<" wynosi: "<<suma(x,y);      //wywołanie funkcji suma z argumentami x i y
    return 0;
}
```

Rys. 4.3. Przykład wywoływania funkcji z dwoma argumentami

Definicja funkcji jest umieszczona przed funkcją główną **main**. Ma ona dwa argumenty typu rzeczywistego i zwraca wartość typu rzeczywistego. Jeśli funkcja wykonuje tylko jedno działanie, to można je zapisać od razu w instrukcji **return**.

Funkcję **suma** wywołujemy przez instrukcję:

```
suma(x,y);
```

Zmienne **x** oraz **y** są zmiennymi globalnymi i są to zmienne typu rzeczywistego, tak jak zmienne **a** oraz **b** będące zmiennymi lokalnymi, czyli rozpoznawalnymi tylko w obrębie funkcji **suma**. Zmienne **a**, **b** służą do zapisania wyrażenia, które funkcja ma wykonać. Funkcja pracuje na zmiennych **a** i **b**, będących kopiami zmiennych **x** i **y**, przekazanych do funkcji jako argumenty.

Jest to przykład **przekazywania argumentów przez wartość**. W momencie, gdy zmienne są przekazywane jako argumenty, funkcja tworzy ich kopie w pamięci. Oznacza to, że wszelkie zmiany na wartościach zmiennych nie będą widoczne w miejscu, gdzie zostały przekazane do funkcji. Ponadto nazwy tych zmiennych mogą, ale nie muszą być takie same jak nazwy argumentów funkcji.

Kopiowanie danych przekazywanych do funkcji bardzo często nie jest pożądanym efektem. Dane przekazywane do funkcji mogą bowiem zajmować dużo miejsca w pamięci i wówczas będziemy niepotrzebnie tracili moc obliczeniową komputera na tworzenie ich kopii. Aby zapobiec kopiowaniu danych, wystarczy użyć symbolu **&** w odpowiednim miejscu argumentu, czyli zastosować **przekazywanie argumentów przez referencję** („przezwisko"). Symbol **&** umieszcza się za typem zmiennej i przed jej nazwą, np.:

```
int &zmienna;
```

Jeśli chcemy przekazywać argumenty przez referencję, stosujemy następujący zapis:

```
typ_funkcji nazwa_funkcji(typ_argumentu &nazwa_arg /*kolejne argumenty*/)
```

Dane przekazane przez referencję nie będą kopiowane. Będziemy pracowali na oryginalnych danych, co w praktyce oznacza, że modyfikacja wartości zmiennej zmodyfikuje nam tę zmienną, która została przekazana do funkcji przez argument.

PRZYKŁAD 4.4

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

void zamien(int &a, int &b)
{
  int x;
  x = a; a = b; b = x;
}

int main()
{
    int x=5;
    int y=6;
cout<<"Przed zmiana: "<<endl;
cout << " x = "<<x<<" y= "<<y<<endl;
zamien(x,y);
cout<<"Po zmianie: "<<endl;
cout << " x = "<<x<<" y= "<<y<<endl;
return 0;
}
```

Rys. 4.4. Przekazywanie argumentów przez referencję

Jeśli definicję funkcji umieścimy po funkcji głównej **main**, to nie będzie ona widoczna dla kompilatora, chyba że przed funkcją **main** umieścimy prototyp naszej funkcji.

Prototyp (deklaracja) funkcji to model, pod którym funkcja będzie rozpoznawalna w programie. W przeciwieństwie do definicji prototyp jest zakończony średnikiem. Definicja funkcji zaś musi mieć nagłówek, szkielet (zawierający wszystkie instrukcje wykonywane w obrębie danej funkcji) oraz instrukcję powrotu (część kończącą każdą funkcję).

Nagłówek definicji funkcji ma budowę identyczną z prototypem, ale nie jest zakończony średnikiem. W przypadku kiedy funkcja ma oddzielną deklarację (prototyp) i definicję, nie jest konieczne nazywanie argumentów funkcji podczas deklaracji. Wystarczą jedynie typy tych argumentów. Często używa się tej właściwości, żeby nie zaciemniać deklaracji funkcji niepotrzebnymi nazwami argumentów.

PRZYKŁAD 4.5

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

float delta(float a, float b, float c)            //funkcja o nazwie delta typu float, pracujaca na argumentach typu float
{
    return b*b-4*a*c;
}

void dwa_rozw(float a, float b, float d)          //funkcja o nazwie dwa_rozw typu void, pracujaca na argumentach typu float
{
    float x1=(-b-sqrt(d))/(2*a);
    float x2=(-b+sqrt(d))/(2*a);
    cout<<"x1= "<<x1<<endl<<"x2= "<<x2;
}
float jedno_rozw(float a,float b)                 //funkcja o nazwie jedno_rozw typu float, pracujaca na argumentach typu float
{
    return -b/a;
}
int main()
{
    float a,b,c;
    cout<<"Rozwiazywanie rownania ax^x+bx+c=0"<<endl;
    cout<<"Podaj a: ";
    cin>>a;
    cout<<"Podaj b: ";
    cin>>b;
    cout<<"Podaj c: ";
    cin>>c;
```

Rys. 4.5. Rozdzielenie deklaracji funkcji od jej definicji

Program może zawierać dowolną liczbę funkcji.

PRZYKŁAD 4.6

```
if (a==0)
{
    cout<<"Rownanie jest liniowe i posiada jedno rozwiazanie: "<<jedno_rozw(b,c);
}
else
{
    float d=delta(a,b,c);                    //wywolanie funkcji delta
    if (d<0)
    {
        cout<<"Rownanie nie posiada rozwiazania";
    }
    else
    {
        if (d==0)
        {
            cout<<"Rownanie posiada jedno rozwiazanie: "<<jedno_rozw(2*a,b);  //wywolanie funkcji jedno_rozw
        }
        else
        {
            cout<<"Rownanie posiada dwa rozwiazania"<<endl;
            dwa_rozw(a,b,d);                                                  //wywolanie funkcji dwa_rozw
        }
    }
}
return 0;
}
```

Rys. 4.6. Zastosowanie wielu funkcji w jednym programie

SPRAWDŹ SWOJĄ WIEDZĘ

1. W jaki sposób wywołuje się funkcję?
2. Czym różni się prototyp od definicji funkcji?

4.3 Przeładowanie funkcji

ZAGADNIENIA
- **Znaczenie przeładowania funkcji**
- **Przykłady przeładowania funkcji**

Przeładowanie nazwy funkcji polega na tym, że w danym programie występuje więcej niż jedna funkcja o takiej samej nazwie. To, która z nich zostanie zastosowana w kodzie, zależy od argumentów, z którymi zostanie ona wywołana.

Rozważmy funkcję do obliczania pola powierzchni wybranych figur. Inaczej będzie wyglądała funkcja obliczająca pole koła, a inaczej – trójkąta. Pisanie kilku funkcji o różnych nazwach dla poszczególnych figur jest trochę pracochłonne. Zamiast nazywać funkcje: **`pole_kola`**, **`pole_trapezu`**, **`pole_trojkata`** itd., wygodniej jest napisać **`pole`**. C++ dopuszcza taką możliwość. Jeżeli w jednym zakresie istnieje kilka funkcji o tych samych nazwach, to należy je jakoś odróżniać. Wystarczy zadbać o to, aby typy argumentów były różne w każdej z funkcji lub by była inna liczba argumentów. Można też zamienić kolejność – błędu wtedy nie będzie.

PRZYKŁAD 4.7

```
float pole(float promien);
float pole(float dl, float szer);
```

Funkcje wyglądają podobnie. Mają jednakowe nazwy, a mimo to kompilator nie wyświetla żadnego błędu. Na tym właśnie polega przeładowanie nazwy funkcji. Takich funkcji może być więcej. Wystarczy zmienić typy argumentów, ich liczbę lub kolejność.

Uwaga!
Podczas przeładowania nazwy funkcji ważna jest **jedynie** odmienność argumentów. Typ zwracany przez funkcję nie jest brany pod uwagę. Zatem niepoprawny będzie następujący sposób przeładowania funkcji:
- float pole (int promień);
- int pole (int promień).

W trakcie kompilacji takie przeładowanie będzie traktowane jako błąd.

PRZYKŁAD 4.8

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

float pole(float promien)          //funkcja obliczająca pole koła, posiada jeden argument typu float
{
    return M_PI*promien*promien;
}

float pole(float a, float b)       //funkcja obliczająca pole prostokata, posiada dwa argumenty typu float
{
    return a*b;
}
int main()
{
    float pr,x,y;
    cout<<"Podaj dlugosc promienia: ";
    cin>>pr;
    cout<<"Pole kola o promieniu "<<pr<<" wynosi: "<<pole(pr)<<endl;
    cout<<"Pole prostokata o bokach 3 i 4.5 wynosi: "<<pole(3,4.5);
    return 0;
}
```

Rys. 4.7. Przeładowanie funkcji

Zostały zdefiniowane dwie funkcje „pole" różniące się liczbą argumentów. Pierwsza z nich posiada jeden argument typu zmiennoprzecinkowego, a druga pracuje na dwóch argumentach typu zmiennoprzecinkowego.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Na czym polega przeładowanie funkcji?
2. Podaj przykład przeładowania funkcji.
3. Omów sposoby przekazywania argumentów do funkcji.

4.4 Argumenty domniemane

ZAGADNIENIA

- Funkcja argumentów domniemanych
- Przykład argumentów domniemanych

Jeśli podczas wywoływania funkcji podamy złą liczbę argumentów, to kompilator wyświetli błąd. W języku C++ w funkcjach można umieszczać argumenty domniemane.

Argument domniemany to argument, który może zostać podany w wywołaniu funkcji lub nie. Argumenty te określa się w deklaracji funkcji, a nie w jej definicji. Jeśli jednak funkcja jest w programie napisana powyżej jakiegokolwiek jej wywołania, to oddzielna deklaracja tej funkcji nie jest potrzebna. Sama definicja funkcji jest wtedy także jej pierwszą deklaracją występującą w tym programie.

Rozpatrzmy funkcję służącą do wyświetlenia informacji o temperaturze. Zazwyczaj temperatura jest podawana w stopniach Celsjusza. Czasem jednak chcemy mieć inną skalę. Rodzaj jednostki można określić w wywołaniu funkcji za pomocą określonej liczby całkowitej. Wartość 0 to stopnie Celsjusza, 1 – kelwiny, a 2 – stopnie Fahrenheita. Wygodnie byłoby mieć funkcję, która w domyśle podaje skalę Celsjusza, a jedynie na specjalne życzenie w stopniach Fahrenheita czy w kelwinach. Jest to możliwe przy zastosowaniu argumentów domniemanych. Jeżeli nie podamy nic, zostanie wykorzystana skala Celsjusza, w przeciwnym przypadku w kelwinach lub stopniach Fahrenheita.

PRZYKŁAD 4.9

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

float temperatura(float stopnie, int skala = 0) ;  //skala jako argument domniemany
{
if (skala == 0) return stopnie;
if (skala == 1) return stopnie+273;
if (skala == 2) return stopnie*1.8+32;
}
int main()
{
    float temp;
    cout<<"Dzisiaj jest: "<<temperatura(26)<<" C"<<endl;        //wywołanie funkcji z arguementem skala=0
    cout<<"Dzisiaj jest: "<<temperatura(26,1)<<" K"<<endl;
    cout<<"Dzisiaj jest: "<<temperatura(26,2)<<" F"<<endl;
    return 0;
}
```

Rys. 4.8. Argumenty domniemane funkcji

Taką funkcję można wywołać dwojako. Pierwszy sposób to normalne wywołanie z określeniem konkretnej wartości argumentu skala:

```
int t = temperatura(26,0);
```

Drugi sposób jest nieco krótszy, ponieważ pomijamy wartość drugiego argumentu:

```
int t = temperatura(26);
```

W obu powyższych zapisach zostanie po prostu wypisana temperatura 26°C. Jeśli chcemy wyświetlić tę temperaturę w kelwinach, należy zastosować zapis:

```
int t = temperatura(26,1);
```

Uwaga!
Wszelkie argumenty domniemane **trzeba** umieszczać na końcu. Nie może być sytuacji, w której pierwszy argument będzie domniemany, a pozostałe nie.

PRZYKŁAD 4.10

Rozważmy zapis:

```
void wypisz(int arg1=0, int arg2, int arg3=1);
```

Jeżeli podczas wywołania funkcji zostaną podane dwa argumenty, to mogą mieć miejsce dwie sytuacje:

a) funkcja zostanie wywołana z pierwszym argumentem domniemanym równym 0, argumenty drugi i trzeci to wartości podane w wywołaniu;
b) argumenty pierwszy i drugi określimy w wywołaniu, a trzeci, jako że jest domniemany, będzie równy 1.

Dlatego taka funkcja nie może zostać wywołana. Kompilator wyświetli komunikat o błędzie.

SPRAWDŹ SWOJE UMIEJĘTNOŚCI

1. Napisz program zawierający funkcję sprawdzającą, czy z trzech liczb podanych przez użytkownika można zbudować trójkąt prostokątny. Niech to będzie funkcja typu **void**, z trzema argumentami typu **float**.
2. Napisz program zawierający cztery funkcje wykonujące cztery podstawowe działania arytmetyczne. Każda z funkcji ma zwracać wartość typu rzeczywistego i pobierać dwa argumenty typu **float**.
3. Napisz program, który pobierze z klawiatury trzy wartości i przypisze je pewnym zmiennym, które przekaże jako argumenty pewnej funkcji. Jeśli wartość pierwszej zmiennej pomnożonej przez wartość drugiej zmiennej jest większa od 100-krotności trzeciej zmiennej, to wynikiem funkcji będzie podwojona wartość pierwszej zmiennej, w przeciwnym razie wynikiem funkcji będzie wartość średnia arytmetyczna trzech zmiennych.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Co to są argumenty domniemane?
2. Podaj przykład zastosowania argumentów domniemanych.

5. Tablice

- Tablice jednowymiarowe
- Przekazywanie tablicy do funkcji
- Tablice wielowymiarowe

5.1 Tablice jednowymiarowe

ZAGADNIENIA

- **Co to jest tablica?**
- **Postać definicji tablicy jednowymiarowej**
- **Przykłady tablic jednowymiarowych**

Tablica to zmienna, która może przechowywać więcej niż jedną wartość tego samego typu. Czas dostępu do każdego z elementów tablicy nie zależy od położenia elementu w tej strukturze. Aby zdefiniować tablicę, należy określić:

- nazwę tablicy;
- typ wartości elementów, które będzie przechowywać tablica;
- liczbę elementów (wymiar tablicy).

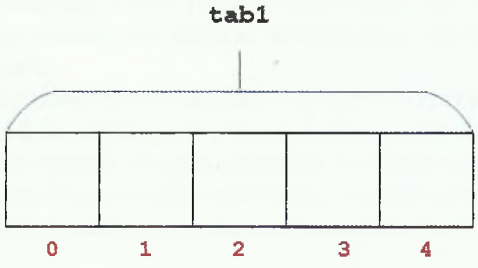
Definicja tablicy jednowymiarowej ma następującą postać:

```
typ nazwa_tablicy[ilość_elementów];
```

PRZYKŁAD 5.1

```
int tab1[5];
```

Jest to definicja pięcioelementowej tablicy o nazwie **tab1** z elementami typu całkowitego.



Rys. 5.1. Postać jednowymiarowej tablicy

Uwaga!
Elementy w tablicy są numerowane od zera. W przypadku pięcioelementowej tablicy pierwszym elementem jest element zerowy, natomiast ostatnim – element czwarty.

Aby wpisać wartość do komórki tablicy, należy podać nazwę tej tablicy, a następnie w nawiasach kwadratowych wpisać indeks komórki.

PRZYKŁAD 5.2

Aby wpisać kolejne liczby naturalne do tablicy **tab1**, możemy zastosować zapis:

```
tab1[0]=1;
tab1[1]=2;
tab1[2]=3;
tab1[3]=4;
tab1[4]=5;
```

Rys. 5.2. Przypisanie wartości w tablicy

Powyższy sposób przypisania wartości poszczególnym elementom tablicy jest poprawny, ale nie jest praktyczny. Zwykle elementy tablicy czyta się lub wypisuje za pomocą pętli **FOR**.

PRZYKŁAD 5.3

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

int main()
{
   int tab1[5];
   for (int i=0;i<5;i++)
   {
       tab1[i]=i+1;
       cout<<"tab["<<i<<"]= "<<tab1[i]<<endl;
   }
    return 0;
}
```

Rys. 5.3. Przypisanie wartości w tablicy za pomocą pętli FOR

Innym sposobem nadania wartości elementom tablicy jest **inicjalizacja**, czyli nadanie wartości początkowych w momencie definiowania tablicy.

PRZYKŁAD 5.4

Rozważmy przypisanie:

```
int tab1[5]={1,2,3,4,5,7};
```

Jeśli między klamrami zostanie umieszczona zbyt duża ilość elementów, to kompilator wyświetli błąd podczas kompilacji. Jeśli zaś zastosujemy zapis:

```
int tab1[5]={1,2};
```

to elementowi zerowemu zostanie przypisana liczba 1, elementowi pierwszemu – 2, natomiast trzy pozostałe zostaną uzupełnione zerami.

PRZYKŁAD 5.5

Przypisz wartości początkowej tablicy 5-elementowej: 1, 2, 5, 0, 0. Wyświetl wartości parzyste tej tablicy oraz wartości nieparzyste.

```
#include <iostream>

using namespace std;


int main()
{
  int tab[5] = {1, 2, 5};

  cout<<"Wartości parzyste: ";
  for(int i=0;i<5;i++)     //w pętli sprawdzamy czy wartość w tablicy jest parzysta
    if(tab[i]%2==0)
      cout<<tab[i]<<" ";
cout<<endl;
cout<<"Wartości nieparzyste: ";
  for(int i=0;i<5;i++)      //w pętli sprawdzamy czy wartość w tablicy jest nieparzysta
    if(tab[i]%2!=0)
      cout<<tab[i]<<" ";
  return 0;
}
```

Rys. 5.4. Wartości parzyste i nieparzyste tablicy

Zmienna **tab** została zainicjalizowana tylko trzema elementami różnymi od zera, ponieważ wiadomo, że pozostałe elementy tablicy zostaną uzupełnione zerami. Pierwsza pętla **FOR** wyszukuje i wyświetla tylko wartości parzyste. Aby sprawdzić parzystość, czyli podzielność przez 2, używa się operatora **%**. Druga pętla **FOR** wypisuje wartości nieparzyste, zatem warunek, który w niej sprawdzamy, jest przeciwny do tego w pętli pierwszej. Żadna z pętli nie potrzebuje klamer otwierającej i zamykającej, ponieważ w ich wnętrzu znajduje się pojedyncza instrukcja.

PRZYKŁAD 5.6

Wylosuj liczby całkowite z zakresu od 1 do 100, wstaw je do 10-elementowej tablicy, a następnie wyświetl te wartości i oblicz ich sumę. Wynik wypisz na ekranie.

```
#include <iostream>
#include <math.h>
#include <cstdlib>
#include <cstdio>
#include <ctime>
using namespace std;

int main()
 {
    int t[10],i,j;
   srand(time(NULL));
    int suma=0;

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

```
        t[i]=rand() % 100  + 1;
        cout<<"t["<<i<<"]= "<<t[i]<<endl;
        suma=suma+t[i];
    }

    cout<<"Suma wartości w tablicy wynosi: "<<suma;
    return 0;
}
```

Rys. 5.5. Zastosowanie funkcji rand()

Funkcja **rand**, losująca liczby całkowite, pochodzi ze standardowej biblioteki **cstdlib**. Funkcja ta losuje nam liczbę całkowitą w przedziale od 0 do RAND_MAX (predefiniowana stała symboliczna, $32767=2^{15}-1$).

Jeżeli uruchomimy kilka razy powyższy program bez użycia funkcji **srand**, to za każdym razem zostaną wylosowane te same liczby. Funkcja **srand** ustawia punkt startowy dla mechanizmu generowania kolejnych liczb całkowitych. Oznacza to, że zapis:

srand(120);
ustawi punkt startowy 120.

Aby uzyskać każdorazowo inne wylosowane wartości w raz skompilowanym programie, należy powiązać funkcję **srand** z czasem rzeczywistym, ustawionym obecnie w komputerze. W tym celu należy dołączyć bibliotekę **ctime**, która umożliwia między innymi skorzystanie z funkcji **time**. Funkcja ta zwraca obecny czas na komputerze w postaci liczby.

Zapis **rand()%100 +1** zainicjuje losowanie liczb od 1 do 100.

Uwaga!
Zapis **srand(time(NULL))** należy wywołać tylko raz na samym początku programu.

Tabela 5.1. Zastosowanie funkcji **rand()**

Zastosowanie funkcji rand() do generowania liczb losowych	
liczba = p + rand()% (q-p+1);	Generowanie liczby losowej całkowitej z zakresu [p;q]
liczba = rand() % (q+1);	Generowanie liczby losowej całkowitej z zakresu [0;q]
liczba = p + (double) rand()/ RAND_MAX*(q-p);	Generowanie liczby losowej rzeczywistej z zakresu [p;q]
liczba = (double) rand()/ RAND_MAX;	Generowanie liczby losowej rzeczywistej z zakresu [0;1]

SPRAWDŹ SWOJĄ WIEDZĘ

1. Co to jest tablica?
2. Podaj postać definicji tablicy jednowymiarowej.
3. Od jakiej cyfry numerowane są elementy w tablicy?
4. Podaj metody nadawania wartości elementom tablicy.

5.2 Przekazywanie tablicy do funkcji

ZAGADNIENIA

- **Sposób przesyłania parametrów do funkcji**
- **Przykłady przekazywania tablicy do funkcji**

Jak już wiadomo, przesyłanie parametru do funkcji odbywa się przez wartość lub przez referencję. Tablicy jednak nie da się przesłać przez wartość. W taki sposób można przekazać pojedynczy element, ale nie całą zawartość tablicy.

Tablice przesyła się, podając funkcji tylko adres początku tablicy.

PRZYKŁAD 5.7

Weźmy pod uwagę funkcję:

```
void funkcja(int tab[]);
```

której argumentem jest tablica liczb całkowitych. Wywołujemy ją w następujący sposób:

```
int t[]={3,2,4,5};
funkcja(t);
```

W nazwie tablicy wywoływanej na potrzeby funkcji nie używamy nawiasów kwadratowych.

Uwaga!

Nazwa tablicy jest równocześnie adresem jej zerowego elementu.

Wyrażenie `tab+3` jest adresem miejsca w pamięci, w którym istnieje element o indeksie 3. Element o indeksie 3, to element `tab[3]`, natomiast adres takiego elementu to `&tab[3]`.

Znak `&` (ampresand) jest jednoargumentowym operatorem uzyskującym adres danego obiektu.

Oznacza, że zapis `tab+3` jest równoważny zapisowi `&tab[3]`.

PRZYKŁAD 5.8

```
#include<iostream>
#include<cstdlib>
using namespace std;
void zsumuj(int tab[])
{
        int suma=0;
        for(int i=0; i<5; i++)
                suma=suma+tab[i];
        cout<<"Suma wynosi: "<<suma;
}
int main()
{
        //inicjacja tablicy
        int tablica[5] = {1,2, 3, 4, 5};
        //wypisanie elementów tablicy
        for(int i=0;i<5;i++)
            cout<<tablica[i]<<" ";
        //wstawienie znaku końca linii (enter)
        cout<<endl;
        //wykonanie polecenia
        zsumuj(tablica); //przekazując tablicę, podajemy tylko jej nazwę
        return 0;
}
```

Rys. 5.6. Przekazywanie tablicy do funkcji

SPRAWDŹ SWOJĄ WIEDZĘ

1. W jaki sposób odbywa się przekazywanie tablicy do funkcji?
2. Przedstaw zależność między nazwą tablicy a adresem jej zerowego elementu.

5.3 Tablice wielowymiarowe

ZAGADNIENIA

- Co to jest tablica wielowymiarowa?
- Przykłady tablic wuielowymiarowych

Tablice wielowymiarowe to tablice, których elementami są inne tablice. Są to po prostu tablice tablic.

Postać deklaracji tablicy dwuwymiarowej:

```
typ nazwa_tablicy[il_elementów][il_elementow];
```

PRZYKŁAD 5.9

```
float tab3[4][2];
```

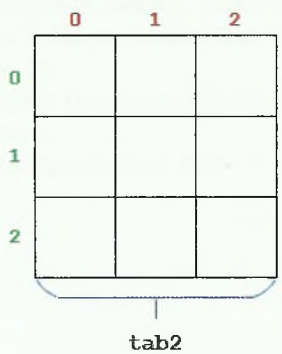
W ten sposób została zdefiniowana czteroelementowa tablica tab3. Każdy z elementów zawiera dwuelementową tablicę liczb typu rzeczywistego.

Aby odwołać się do każdej z komórek, należy w nawiasach kwadratowych podać numer wiersza i numer kolumny komórki, do której się odwołujemy. Trzeba pamiętać o tym, że numerujemy je od zera.

PRZYKŁAD 5.10

Załóżmy, że zadeklarowaliśmy zmienną **tab2**, która jest tablicą dwuwymiarową o trzech wierszach i trzech kolumnach, czyli:

```
float tab2[3][3];
```



Rys. 5.7. Postać tablicy dwuwymiarowej

Aby wpisać wartości do tej tablicy, można zastosować zapis:

```
tab2[0][0]=1;
tab2[0][1]=2;
tab2[0][2]=3;
tab2[1][0]=4;
tab2[1][1]=5;
tab2[1][2]=6;
tab2[2][0]=7;
tab2[2][1]=8;
tab2[2][2]=9;
```

Rys. 5.8. Nadawanie wartości w tablicy dwuwymiarowej

Oczywiście jest to sposób niewygodny i mało czytelny. Zwykle elementy tablicy dwuwymiarowej wczytuje się lub wypisuje za pomocą podwójnej pętli **FOR**.

PRZYKŁAD 5.11

```
float tab2[3][3];

for (int i=0;i<3;i++)
{
    for (int j=0;j<3;j++)
    {
        cout<<"Podaj wartość elementu [" <<i<<"]["<<j<<"]: ";
        cin>>tab2[i][j];
    }
}
```

Rys. 5.9. Zastosowanie podwójnej pętli do wprowadzenia wartości w tablicy dwuwymiarowej

Powyższy kod poprosi użytkownika o podanie wartości dla kolejnych elementów tablicy. Wprowadzane wartości będą uzupełniane wierszami,

czyli w pierwszym wierszu są to elementy: `[0][0]`, `[0][1]`, `[0][2]`,

potem – w drugim wierszu: `[1][0]`, `[1][1]`, `[1][2]`,

a w ostatnim – elementy: `[2][0]`, `[2][1]`, `[2][2]`.

Wprowadzone wartości mogą zostać wyświetlone za pomocą tych samych pętli **FOR**. Można również użyć oddzielnych instrukcji w celu wyświetlenia tablicy w postaci macierzowej. Dzięki temu kod programu staje się czytelniejszy, ponieważ rozdzielamy w ten sposób wprowadzanie wartości do tablicy od ich wyświetlenia.

PRZYKŁAD 5.12

```
#include <math.h>
#include <cstdlib>
#include <cstdio>
#include <ctime>
using namespace std;
int main()
{
    int tab2[3][3];
    int i,j;
    srand(time(NULL));

    for ( i=0;i<3;i++)
    {
        for ( j=0;j<3;j++)
        {
            tab2[i][j]=rand() % 10 +1;      //losowanie wartości z zakresu od 1 o 10
        }
    }


    for (i=0;i<3;i++)
    {
        cout<<endl;
        for(j=0;j<3;j++)
        {
        cout<<tab2[i][j]<<"\t";
        }
    }
    return 0;
}
```

Rys. 5.10. Wyświetlanie wartości tablicy dwuwymiarowej w postaci macierzowej

Powyższy program wygeneruje liczby od 1 do 10 i wstawi je do tablicy **tab2**, następnie wypisze wpisane liczby w postaci macierzowej:

```
8       2       5
6       1       5
5       3       1
Process returned 0 (0x0)   execution time : 0.000 s
Press any key to continue.
```

Rys. 5.11. Przykład wyświetlania wartości tablicy dwuwymiarowej w postaci macierzowej

Wartości początkowe tablicy możemy nadać także podczas jej deklaracji:

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

Poszczególne wiersze tablicy są wyróżnione nawiasami i klamrowymi. Powyższy zapis wypełni tablicę w następujący sposób:

	0	1
0	1	4
1	4	5

Rys. 5.12. Wypełniona tablica dwuwymiarowa

Nawiasy klamrowe można pominąć.

```
int b[2][3]={3,6,2,4,1,0};
```

Ponadto podczas nadawania wartości tablicy można pominąć wartość w pierwszym nawiasie kwadratowym:

```
float tablica[][2] = {{1,1},{4,5},{0,-1}};
```

Deklaracja tablic wyższych wymiarów odbywa się tak samo. Zapis lub odczyt wartości przechowywanych w tablicy wielowymiarowej odbywa się na tych samych zasadach, co dla tablicy jednowymiarowej. Należy odpowiednio podać „współrzędne" danego elementu w tablicy.

SPRAWDŹ SWOJE UMIEJĘTNOŚCI

1. Napisz program obliczający i wyświetlający średnią arytmetyczną wartości 8-elementowej tablicy. Następnie sprawdź, czy wynik jest liczbą większą, mniejszą czy równą 5, i wypisz odpowiedni komunikat.
2. Napisz program, który wylosuje liczby z zakresu od 1 do 500, wstawi je do 10-elementowej tablicy, a następnie wyświetli wartości, które są jednocześnie parzyste i podzielne przez 3.
3. Napisz program, który do 20 elementowej tablicy wczyta wylosowane liczby z zakresu od 10 do 50, a następnie wypisze wartość najmniejszą i wartość największą.
4. Napisz program, który posortuje rosnąco elementy 10-elementowej tablicy. Wartości elementów zostaną wczytane z klawiatury.
5. Napisz program, który wyszuka i wyświetli na ekranie maksymalny element tablicy o wymiarze 3 × 3.
6. Napisz program, który obliczy i wyświetli na ekranie sumę elementów na głównej przekątnej tablicy o wymiarach 3 × 3.
7*. Napisz program obliczający sumę elementów w wierszach tablicy dwuwymiarowej o trzech wierszach i trzech kolumnach.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Podaj postać deklaracji tablicy dwuwymiarowej.
2. W jaki sposób nadaje się wartości elementów tablicy dwuwymiarowej?
3. Dlaczego warto stosować instrukcję FOR do nadawania wartości elementów tablicy?

* Zadanie ponadprogramowe.

6.1 Inicjalizacja łańcucha znaków

ZAGADNIENIA
- Co to jest string?
- Inicjalizacja łańcucha znaków

Łańcuch znaków (ang. *string*) to ciąg złożony z zera lub większej liczby znaków zawartych między znakami cudzysłowu, np. „kot". W rzeczywistości łańcuch znaków jest tablicą, której elementami są pojedyncze znaki (tablica elementów typu char). Ostatnim elementem tablicy jest znak o kodzie 0 (stała liczbowa 0 lub stała znakowa '\0'), oznaczający koniec napisu.

Uwaga!
Rozmiar fizycznej pamięci przeznaczonej na napis musi być o jeden większy niż liczba znaków zawartych między znakami cudzysłowu.

Deklaracja zmiennej mogącej przechowywać napis jest podobna do deklaracji zwykłej tablicy:

```
char nazwa_tablicy[ilość_elementow];
```

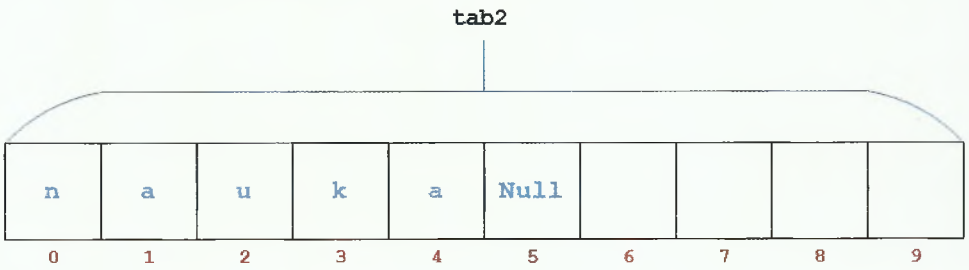
PRZYKŁAD 6.1

```
char tab2[10];
```

Zmienna **tab2** jest tablicą zawierającą do 9 znaków.

Deklarowanemu łańcuchowi znaków możemy nadać wartość początkową:

```
char tab2[10]={"nauka"};
```



Rys. 6.1. Inicjalizacja zmiennej typu string

Napis umieszczamy w cudzysłowie. W rezultacie każda komórka tablicy **tab2** zawiera kolejną literę, komórka o indeksie 5 zawiera wartość NULL jako koniec tekstu, pozostałe elementy są uzupełnione zerami.

To samo można uzyskać przez podanie pojedynczych liter umieszczonych w apostrofach:

```
char tab2[10]={'n','a','u','k','a'};
```

Jeśli zastosujemy zapis:

```
char tab2[]={"nauka"};
```

to kompilator policzy, ile liter ma podany tekst (w tym przypadku 5). W związku z tym, że tekst został ujęty w cudzysłów, zostanie zarezerwowany jeszcze jeden element na znak NULL.

Powyższe sposoby nadawania wartości łańcuchowi znaków są poprawne tylko przy deklaracji. Błędnym jest zatem poniższy zapis:

```
char tekst[10];
tekst = „Pies”;
```

Poprawny zapis wymaga wykorzystania funkcji **strcpy()** z pliku nagłówkowego **string.h**:

```
char txt[10];
strcpy(txt,"Pies");
```

SPRAWDŹ SWOJĄ WIEDZĘ

1. Co to jest łańcuch znaków?
2. Wymień sposoby inicjalizacji tablicy znaków.

6.2 Podstawowe funkcje działające na tablicach znaków

ZAGADNIENIA

- **Funkcje działające na łańcuchach**
- **Zastosowanie funkcji działających na łańcuchach**

Aby wprowadzić na ekran zawartości łańcuchów, można zastosować znaną instrukcję **printf**, natomiast w celu odczytania tekstu podanego z klawiatury można użyć instrukcji **scanf**.

Dodatkowo można korzystać z funkcji bibliotecznych **puts** i **gets**. Służą one do wpisywania i wczytywania łańcuchów i nie wymagają kodów formatujących. Argumentem funkcji **puts** może być stała łańcuchowa lub nazwa tablicy, która przechowuje łańcuch.

Funkcja **gets** odczytuje łańcuch z klawiatury i zapamiętuje go w tablicy będącej argumentem funkcji. Funkcja odczytuje również łańcuchy zawierające spacje.

PRZYKŁAD 6.2

```
char imie[20];
puts(„Podaj swoje imie: „);
gets(imie);
```

W bibliotece **<strin.h>** zdefiniowano wiele funkcji służących do przetwarzania danych tekstowych. Przykładowe funkcje działające na łańcuchach znaków zebrano w tablicy 6.1.

Tabela 6.1. Funkcje działające na łańcuchach.

Nazwa funkcji	Nagłówek funkcji	Działanie funkcji
strlen()	int strlen(s);	Funkcja zwraca długość łańcucha znaków; nie bierze pod uwagę znaku **'\0'**.
strcpy()	char strcpy(s1, s2);	Funkcja kopiuje łańcuch **s2** do łańcucha **s1**.
strcat()	char strcat(s1, s2);	Funkcja dołącza do łańcucha **s1** łańcuch **s2**.
strchr()	char strchr(s, int c);	Funkcja przeszukuje łańcuch **s** w celu znalezienia pierwszego wystąpienia znaku c; jeśli znak nie został znaleziony, to zwraca wskaźnik do znalezionego znaku lub **NULL**.

Nazwa funkcji	Nagłówek funkcji	Działanie funkcji
strcmp()	int strcmp(s1, s2);	Funkcja porównuje łańcuchy **s1** i **s2** z rozróżnianiem wielkości liter, zwraca **0**, gdy **s1=s2**, wartość mniejszą od zera, gdy **s1<s2** i wartość większą od zera, gdy **s1>s2.**
strcmpi()	int strcmpi(s1, s2);	Funkcja działa jak **strcmp()**, ale bez rozróżniania wielkości liter;
strlwr()	char strlwr(s);	Funkcja zamienia w łańcuchu **s** duże litery na małe.
strrev()	char strrev(s);	Funkcja odwraca kolejność znaków w łańcuchu **s**.
strset()	char strset(s, c);	Funkcja wypełnia łańcuch **s** znakiem **c**.
strstr()	char *strstr(s1, s2);	Funkcja przeszukuje łańcuch **s1** w celu odnalezienia podłańcucha **s2** zwracając wskaźnik do elementu łańcucha **s1**, od którego zaczynają się znaki takie same jak w łańcuchu **s2**, lub **NULL**, gdy **s2** nie występuje w **s1**.
strupr()	char strupr(s);	Funkcja zamienia w łańcuchu s litery małe na duże.

PRZYKŁAD 6.3

```
#include <stdio.h>
#include <string.h>
int main()
{
char napis1[] = "Systemy operacyjne";
char napis2[20];
int dlugosc;
printf("napis1: %s \n",napis1);
dlugosc = strlen(napis1);
printf("liczba znakow w napis1: %d \n",dlugosc);
strupr(napis1);
printf("napis1 (duze litery): %s \n",napis1);
strlwr(napis1);
printf("napis1 (male litery): %s \n",napis1);
strcpy(napis2,napis1);
printf("napis2: %s \n",napis2);
strrev(napis2);
printf("napis2 (odwrocony): %s \n",napis2);
return 0;
}
```

Rys. 6.2. Zastosowanie funkcji działających na łańcuchach

Zmienna **napis1** ma wartość początkową, natomiast zmienna **napis2** jej nie ma. Do wypisania tekstu (łańcucha) na ekranie została użyta funkcja **printf**. Aby wyświetlić zawartość tablicy znakowej, używa się parametru **%s**, a do wyświetlenia pojedynczego znaku – parametru **%c**.

Powyższy program ilustruje działanie funkcji operujących na tablicach znaków. Efekt jest następujący:

```
napis1: Systemy operacyjne
liczba znakow w napis1: 18
napis1 (duze litery): SYSTEMY OPERACYJNE
napis1 (male litery): systemy operacyjne
napis2: systemy operacyjne
napis2 (odwrocony): enjycarepo ymetsys
```

Rys. 6.3. Wynik działania programu wykonującego operacje na łańcuchach

PRZYKŁAD 6.4

Napisz program, który obliczy liczbę liter „a" występujących w tekście „System operacyjny to zbior programow do zarzadzania sprzetem komputerowym":

```
#include <string.h>
#include <iostream>
using namespace std;
int main()
{
char napis1[] = "System operacyjny to zbior programow do zarzadzania sprzetem komputerowym";
int dlugosc,ilosc,i;
ilosc=0;
dlugosc = strlen(napis1);

    for (i=0;i<dlugosc;i++)
    {
        if (napis1[i]=='a')
        ilosc++;
    }
cout<<"Ilosc liter o w tekscie: "<<ilosc;
return 0;
}
```

Rys. 6.4. Program obliczający liczbę liter „a" w tekście

SPRAWDŹ SWOJE UMIEJĘTNOŚCI

1. Napisz program, który wyświetli tekst w postaci rozstrzelonej wprowadzony przez użytkownika.
2. Napisz program wypisujący z podanego 10-znakowego tekstu podciąg od drugiego znaku do 8.
3. Napisz program, który sprawdzi, czy podane słowo jest palindromem.
4. Napisz program, który wypisze wszystkie anagramy podanego pięcioliterowego słowa.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Omów funkcje działające na łańcuchach.

7. Wskaźniki

- Definiowanie wskaźników
- Przypisywanie wartości za pomocą wskaźnika
- Zastosowanie wskaźników

7.1 Definiowanie wskaźników

ZAGADNIENIA
- Co to jest wskaźnik?
- Definicja wskaźnika

Wskaźniki mają szerokie zastosowanie w C++, co daje temu językowi ogromną elastyczność. Wskaźnik (ang. *pointer*) – typ zmiennej odpowiedzialnej za przechowywanie adresu do innej zmiennej (innego miejsca w pamięci) w obrębie naszej aplikacji.

Wskaźnik może wskazywać na jakąś zmienną, strukturę, tablicę, a nawet funkcję.

Podstawowe operatory niezbędne do operowania wskaźnikami

- * – operator wyłuskania wartości zmiennej, na którą wskazuje wskaźnik (wyciąga wartość ze wskaźnika);
- & – operator pobrania adresu danej zmiennej, tablicy, struktury (pobiera adres zmiennej).

Wskaźnik zajmuje zazwyczaj 4 bajty bez względu na to, jaki typ danych wskazuje. Rozmiar wskaźnika może być jednak różny w zależności od użytego kompilatora. Wskaźnik zwraca adres pierwszego bajta danych wybranej zmiennej.

Aby pobrać adres dowolnej zmiennej, wystarczy napisać: `&nazwa_zmiennej`.

Aby utworzyć zmienną wskaźnikową, po typie zmiennej trzeba dopisać `*`(gwiazdkę).

PRZYKŁAD 7.1

```
int *liczba;
```

`Liczba` jest wskaźnikiem do pokazywania na obiekt typu `int`. Treścią wskaźnika jest informacja o tym, gdzie wskazywany obiekt się znajduje, a nie, co w nim się znajduje.

Można definiować wskaźniki do obiektów różnych typów, np. `char`, `float`, `int`:

```
char *liczba2;
float *liczba3;
```

Uwaga!

Należy pamiętać, że wskaźnik służący do pokazywania na obiekty jednego typu nie nadaje się do pokazywania na obiekty innego typu.

PRZYKŁAD 7.2

```
#include <iostream>
using namespace std;
int main()
{
    int *wsk;
    int liczba=3;
    wsk=&liczba;
    cout<<"Wsk = "<<*wsk<<endl;
    cout<<"liczba = "<<liczba;
    return 0;
}
```

Rys. 7.1. Zastosowanie wskaźników

W powyższym programie zostały zdefiniowane wskaźnik **wsk** do zmiennej typu całkowitego oraz zmienna całkowita **liczba**. Zapis **wsk=&liczba** oznacza ustawienie wskaźnika na obiekt **liczba**. Kiedy wskaźnik już pokazuje na określone miejsce, możemy odnieść się do tego obiektu, który pokazuje (odczytać jego zawartość). Do takiej operacji służy operator *. Gwiazdka kieruje do obiektu pokazywanego przez wskaźnik. Wskaźnik **wsk** wskazuje zmienną **liczba**. Skoro zawartością **liczba** jest wartość 3, to oba komunikaty **cout** są równoważne.

Wskaźniki stosuje się, gdy zależy nam na:

- ulepszeniu pracy z tablicami;
- funkcjach mogących zmieniać wartość przesyłanych do nich argumentów;
- dostępie do specjalnych komórek pamięci;
- rezerwacji obszarów pamięci.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Co to jest wskaźnik?
2. W jakich przypadkach należy stosować wskaźniki?

7.2 Przypisywanie wartości za pomocą wskaźnika

ZAGADNIENIA

- **Podstawowe operatory wykorzystywane w działaniach na zmiennych wskaźnikowych**

Podstawowymi operatorami wykorzystywanymi w działaniach na zmiennych wskaźnikowych są operator pobrania adresu **&** i operator odwołania do zmiennej wskazywanej *. Jeśli **x** jest zmienną, to **&x** jest adresem tej zmiennej. Operator * umożliwia dostęp do obiektu wskazywanego przez wskaźnik.

PRZYKŁAD 7.3

```
#include <iostream>

using namespace std;

int main()
{
    int  wart =25;
    int *wsk=&wart;
    cout <<"*wsk: "<< *wsk << endl;        //wyświetlenie wyłuskanej wartości wskaźnika (25)
    cout <<"wsk: "<< wsk << endl;          //wyświetlenie adresu zmiennej wart
    cout <<"&wsk: "<< &wsk << endl;        //wyświetlenie adresu wskaźnika
    cout <<"&wart: "<< &wart << endl;      //wyświetlenie adresu zmiennej wart

return 0;
}
```

Rys. 7.2. Przypisanie wartości za pomocą wskaźnika

Wynikiem działania programu jest wyłuskanie wartości wskaźnika oraz adresów zmiennej **wart** oraz adresu wskaźnika.

```
*wsk: 25
wsk: 0x6afefc
&wsk: 0x6afef8
&wart: 0x6afefc

Process returned 0 (0x0)   execution time : 0.020 s
Press any key to continue.
```

Rys. 7.3. Wyłuskanie wartości wskaźnika oraz adresu i wskaźnika

SPRAWDŹ SWOJĄ WIEDZĘ

1. Wymień podstawowe operatory wykorzystywane w działaniach na zmiennych wskaźnikowych.

7.3 Zastosowanie wskaźników

ZAGADNIENIA

- Funkcje wskaźników
- Użycie wskaźników jako argumentów funkcji

Funkcje wskaźników

Za pomocą wskaźników można kontrolować wartości innych zmiennych.

PRZYKŁAD 7.4

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

int main()
{
  int *a, *b, liczba;

  liczba = 5;
  a = &liczba;                          //zmienna a wskazuje na adres zmiennej liczba
  b = &liczba;                          //zmienna b wskazuje na adres zmiennej liczba

  cout<<*a<<" "<<*b<<endl;              //wypisanie wartości, którą przechowuje zmienna liczba

  *a = 7;                               //zmiana wartosci zmiennej liczba

  cout<<*a<<" "<<*b<<" "<<liczba;       //zostanie wypisana trzy razy wartość 7

  return 0;
}
```

Rys. 7.4. Zastosowanie wskaźników

Z rozdziału o tablicach wiadomo, że nazwa tablicy wskazuje na jej pierwszy element. Oznacza to, że możemy odnieść się to tego elementu za pomocą nazwy tablicy i indeksu o wartości 0 lub przez ten wskaźnik.

Po elementach tablicy można poruszać się za pomocą indeksów – podawać kolejne numery komórek, począwszy od 0, jak również przeskakiwać kolejne komórki tablicy z wykorzystaniem wskaźnika.

PRZYKŁAD 7.5

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

using namespace std;

int main()
{
  int tab[6] = {2, 4 ,6, 8, 10, 12};
  cout<<*tab<<endl;                    //wypisanie pierwszego elementu tablicy
  cout<<*(tab+2)<<endl;                //wypisanie trzeciego elementu tablicy
  for(int i=0;i<6;i++)
    cout<<*(tab+i)<<" ";               //wypisanie kolejno elementów tablicy
                                       //poruszając się po niej wskaźnikiem
    cout<<endl;
  return 0;
}
```

Rys. 7.5. Zastosowanie wskaźników dla tablic

Efekt działania powyższego programu będzie następujący:

```
2
6
2 4 6 8 10 12

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

Rys. 7.6. Wynik działania programu ilustrującego działania na tablicach przy pomocy wskaźników

Użycie wskaźników jako argumentów funkcji

Przekazanie wskaźników jako argumentów zapewnia, że każda zmiana wartości zmiennej wewnątrz funkcji będzie „widoczna" w miejscu, w którym została wywołana.

PRZYKŁAD 7.6

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

void zamien(int *liczba1, int *liczba2)
{
    int pom = *liczba1;
        *liczba1 = *liczba2;
        *liczba2 = pom;
}
```

```
int main()
{
        int a, b;
        cin>>a>>b;

        zamien(&a,&b);          //przekazujemy adresy zmiennych

        cout<<a<<" "<<b;        //wartości zmiennych zostały zamienione

        return 0;
}
```

Rys. 7.6. Zastosowanie wskaźników jako argumentu funkcji

Użycie zwykłych zmiennych zamiast wskaźników nie spowoduje zamiany wartości, ponieważ funkcja będzie pracować na kopiach tych zmiennych.

SPRAWDŹ SWOJE UMIEJĘTNOŚCI

1. Napisz program, w którym zadeklarujesz wskaźnik do zmiennej oraz przypiszesz mu odpowiedni adres. Wypisz na ekranie wartość zmiennej przez wskaźnik oraz wypisz adres zmiennej.
2. Napisz funkcję, która posortuje liczby zapisane w zmiennych *a*, *b* i *c*, nadając im kolejność rosnącą. Użyj wskaźników.
3. Napisz program, który zsumuje wartości wszystkich elementów 10-elementowej tablicy Skonstruuj odpowiednią funkcję i zastosuj wskaźniki.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Omów zastosowanie wskaźników.
2. W jaki sposób przekazuje się wskaźniki jako argumenty funkcji?

8. Operacje na plikach

- Otwieranie pliku do odczytu
- Zapisywanie i odczytywanie danych z pliku

8.1 Otwieranie pliku do odczytu

ZAGADNIENIA
- Etapy przetwarzania pliku
- Metoda open
- Odczytywanie danych z pliku

Do przetwarzania plików w języku C++ są stosowane strumienie zrealizowane w postaci następujących klas:
- **ofstream** – klasa umożliwiająca zapis do pliku;
- **ifstream** – klasa umożliwiająca odczytywanie pliku;
- **fstream** – klasa umożliwiająca zapis i odczytywanie pliku.

Narzędzia do pracy z plikami znajdują się w nagłówku **fstream** (**fstream** pochodzi od angielskich słów *file stream* oznaczających 'strumień plikowy'), który dołączamy na początku programu za pomocą dyrektywy **#include <fstream>**. Nazwy zadeklarowane w tym pliku nagłówkowym wchodzą w skład przestrzeni nazw **std**.

Operacje związane z przetwarzaniem pliku składają się z następujących etapów:
1. zdefiniowanie strumienia, czyli stworzenie obiektu jednej z klas: **ifstream**, **ofstream**, **fstream**;
2. otwarcie pliku;
3. wykonanie operacji na pliku;
4. zamknięcie pliku.

Otwarcie pliku do odczytu wykonuje się metodą **open**, gdzie pierwszym argumentem metody jest ścieżka do pliku, który chcielibyśmy otworzyć. Załóżmy, że na dysku `C:\` istnieje plik, który nosi nazwę **dane.txt**. Otwarcie tego pliku wykonają następujące instrukcje:

```
ifstream plik;
plik.open( „C:\\odczyt.txt" );
```

Jeżeli dany plik będzie istniał na dysku oraz nie będzie on zablokowany do odczytu przez inną aplikację, wówczas otwarcie pliku zakończy się powodzeniem.

Otwarcie pliku może zakończyć się zarówno powodzeniem, jak i fiaskiem. Zanim zaczniemy pracować na danych z pliku, warto sprawdzić, czy udało się otworzyć plik. W tym celu stosuje się zazwyczaj metodę **good**, należącą do klasy **ifstream**.

Ścieżka do pliku może być **względna** (określająca położenie pliku względem miejsca, w którym znajduje się program) lub **bezwzględna** (odnosząca się do katalogu głównego na dysku).

PRZYKŁAD 8.1

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

using namespace std;

int main()
{
   ifstream plik;
   plik.open("C:/a/dane.txt");
    if(plik.good())
        {
        cout<<"Udało się otworzyć plik";
        }
    else
        {
        cout<<"Nie można otworzyć pliku do odczytu";
        }
    plik.close(); //zamknięcie pliku
    return 0;
}
```

Rys. 8.1. Otwarcie pliku do odczytu

Odczytywanie danych z pliku

Aby odczytać dane z pliku, należy użyć strumienia **ifstream** w następujący sposób:

```
ifstream mojStrumien(„C:/a/dane.txt");
```

Instrukcja ta spowoduje umieszczenie kursora na początku.

Plik można odczytać na trzy sposoby:

a) linijka po linijce, z wykorzystaniem funkcji **getline()**;

b) słowo po słowie, z wykorzystaniem operatora **>>**;

c) znak po znaku, za pomocą funkcji **get()**.

Pierwszy sposób polega na odczytaniu jednej linijki tekstu i zapisaniu jej jako łańcucha znaków:

```
string tekst;
getline(mojStrumien,tekst);
```

Działanie tej instrukcji jest takie samo jak w przypadku instrukcji **cin**.

W przypadku użycia operatora **>>** jest odczytywane wszystko, co znajduje się między miejscem w pliku, w którym aktualnie znajduje się kursor, a najbliższą spacją. Odczytany tekst jest traktowany jako typ **double**, **int** lub **string**, w zależności od typu użytej zmiennej. Zapis:

```
string tekst2;
mojStrumien >> tekst2;
```

spowoduje, że w zmiennej **tekst2** zostanie zapisane słowo „Plik".

PRZYKŁAD 8.2

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

int main()
{
    ofstream plik("C:/a/dane.txt");
    if(plik)
    {
        string tekst;      // Zmienna do przechowywania odczytanych wierszy tekstu

         cout << tekst << endl; // Wyświetlamy odczytany tekst w konsoli

    }
    else
    {
        cout << "BŁĄD: Nie mozna otworzyc pliku do odczytu." << endl;
    }

    return 0;
}
```

Rys. 8.2. Odczytanie danych z pliku tekstowego

Powyższy kod otwiera plik o nazwie dane.txt. Jeśli plik istnieje, to zostanie wyświetlona jego zawartość. W przeciwnym wypadku zostanie wyświetlony komunikat o błędzie odczytu.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Wymień operacje związane z przetwarzaniem pliku.
2. Omów proces otwierania pliku do odczytu.

8.2 Zapisywanie danych do pliku

ZAGADNIENIA

- Zapisywanie danych do pliku

Zapisywanie danych do pliku

Aby zapisać dane do pliku, należy stworzyć strumień wyjściowy, czyli umożliwiający otwarcie pliku do zapisu. W tym celu trzeba zdefiniować zmienną typu **ofstream: ofstream plik ("ścieżka–do pliku").**

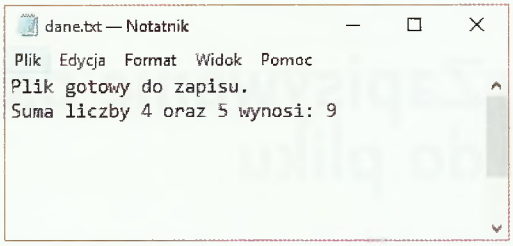
W czasie otwierania pliku kompilator może napotkać różnego rodzaju problemy, np. plik nie zostanie odnaleziony albo dysk twardy będzie zapełniony. Warto więc użyć instrukcji warunkowej w celu sprawdzenia, czy operacja otwarcia pliku się powiodła. Jeśli plik jest gotowy do zapisu, wykorzystujemy strumień **`c_str`** służący do zapisu danych w pliku.

PRZYKŁAD 8.3

```
#include <string>
using namespace std;
int main()
{
    string const nazwaPliku("c:/a/dane.txt");
    ofstream mojStrumien(nazwaPliku.c_str());        // Definicja strumienia do zapisu danych w pliku.
    if(mojStrumien)                                  // Sprawdzenie czy plik zostal otwarty
    {
        mojStrumien << "Plik gotowy do zapisu." << endl;
        int x,y,z;
        x=4;
        y=5;
        z=x+y;
        mojStrumien << "Suma liczby " << x << " oraz " << y <<" wynosi: "<<z<<endl;
    }
    else
    {
        cout << "BŁĄD: Nie mozna otworzyc pliku." << endl;
    }
    mojStrumien.close();     //zamknięcie pliku
    return 0;
}
```

Rys. 8.3. Zapisywanie danych do pliku

Po uruchomieniu tego programu na dysku zostanie utworzony plik o nazwie **`dane.txt`** o następującej zawartości:



Rys. 8.4. Wynik działania programu zapisującego dane do pliku

Jeśli nie chcemy utracić wcześniej zapisanych danych albo kasować za każdym razem zawartości pliku, tylko dodawać na jego końcu coraz to nowe wiersze danych, musimy zastosować poniższy zapis:

```
ofstream mojStrumien(C:/a/dane.txt", ios:app);
```

Słowo **app** (z ang. ***append***) oznacza 'dołącz'. Od tej pory nowe dane nie będą powodowały skasowania starych, tylko będą dodawane na końcu pliku.

Ostatnia z metod polega na odczytywaniu z pliku znak po znaku:

```
char znak;
mojStrumien.get(znak);
```

Powyższy zapis odczyta jeden znak i zapisze go w zmiennej **znak**.

Do odczytywania pliku warto zastosować funkcję **getline()**. Funkcja ta wczytuje linie zawartości plików, ale dodatkowo zwraca wartość logiczną informującą, czy można kontynuować odczyt. Jeśli funkcja zwróci wartość **true**, to znaczy, że plik nie został jeszcze w całości odczytany i można kontynuować odczytywanie. Jeśli zaś zwróci **false**, to znaczy, że został osiągnięty koniec pliku albo wystąpił błąd. Aby odczytać zawartość pliku do końca, należy użyć pętli.

PRZYKŁAD 8.4

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

using namespace std;

int main()
{
    ifstream plik("C:/a/dane.txt");
    if(plik)
        {
            string tekst;
                while(getline(plik,tekst))
                {
                 cout<<tekst<<endl;
                }
        }
    else
        {
        cout<<"Nie można otworzyć pliku do odczytu";
        }
    return 0;
}
```

Rys. 8.5. Odczyt danych z pliku

Odczytywanie pliku będzie odbywało się linijka po linijce, aż dotrzemy do linii pustej.

Po wczytaniu zawartości pliku do łańcucha znaków można na niej pracować z wykorzystaniem funkcji operujących na stringach.

Czasami jednak trzeba zamknąć plik wcześniej, nie czekając na zamknięcie automatyczne. W takim przypadku należy użyć funkcji **close()**.

PRZYKŁAD 8.5

Załóżmy, że na dysku C: w folderze a znajduje się plik tekstowy *dane.txt* z zapisanymi w nim dwiema liczbami 23 i 7. Chcemy odczytać te liczby i wykonać na nich działanie dodawania.

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

using namespace std;

int main()
{
  int x, y;
  ifstream odczyt("c:/a/dane.txt");

  odczyt>>x>>y;

  cout<<"Suma wczytanych liczb "<<x<<" oraz " <<y<<" wynosi: "<<x+y;
  cin.get();
  odczyt.close();        //zamknięcie pliku
  return 0;
}
```

Rys. 8.6. Odczyt danych z pliku i wykonywanie na nich obliczeń

SPRAWDŹ SWOJE UMIEJĘTNOŚCI

1. Napisz program, który pobierze z klawiatury wartości 20 liczb, a następnie policzy ich średnią arytmetyczną. Wynik wpisz do pliku **srednia.txt**.
2. Do pliku o nazwie dane.txt wczytaj trzy liczby całkowite **a**, **b**, **c**. Liczby te wykorzystaj jako współczynniki równania kwadratowego $ax^2+bx+c=0$. Wyniki rozwiązanego równania wpisz do pliku **rozwiązanie.txt**.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Omów zapisywanie danych do pliku.

9. Programowanie obiektowe

- Definicja klasy
- Klasa, obiekt i funkcje w klasach
- Konstruktor i destruktor
- Tablice obiektów klasy
- Dziedziczenie
- Polimorfizm

9.1 Definicja klasy

ZAGADNIENIA
- Co to jest klasa?
- Dane składowe i metody
- Słowa kluczowe

Klasa grupuje pewne zmienne i funkcje w jedną całość. Klasa to inaczej typ danych. Klasy tworzy się ze słowa kluczowego **class**, nazwy klasy oraz ciała klasy zawartego w klamrach. Przykładowa definicja klasy wygląda następująco:

```
class student
{
//ciało klasy
};
```

Rys. 9.1. Przykładowa definicja klasy

W ciele klasy deklarujemy składniki klasy, czyli funkcje i dane. Mogą nimi być różnego rodzaju dane. Nazywamy je danymi składowymi klasy

PRZYKŁAD 9.1

```
class student
{
public:
string imie;
string nazwisko;
int wiek;
};
```

Rys. 9.2. Definicja klasy

Zmienne zdefiniowane wewnątrz klasy nazywamy polami, natomiast funkcje składowe – metodami. Klasa w C++ jest bardzo podobna do struktury. Struktury to złożone typy danych pozwalające przechowywać różne informacje. Za pomocą struktur możliwe jest grupowanie wielu zmiennych o różnych typach w jeden obiekt.

Struktury tworzy się za pomocą słowa kluczowego **struct**:

```
struct nazwa_struktury{
pole danych;
pole danych;
};
```

PRZYKŁAD 9.2

```
struct osoba {
string imie;
string nazwisko;
int wiek;
};
```

Po zdefiniowaniu struktury możemy na jej podstawie tworzyć zmienne strukturalne jak każde inne zmienne. Typem będzie nazwa struktury:

```
nazwa_struktury  nazwa_zmiennej;
```

Aby odwołać się do zmiennych struktury należy podać **nazwę obiektu struktury**, następnie – **po kropce – nazwę pola**.

PRZYKŁAD 9.3

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

using namespace std;

struct student{              // deklaracja struktury
    string imie;
    string nazwisko;
    int wiek;
};

int main()
{
    student uczl;              //tworzenie obiektu struktury o nazwie uczl

    uczl.imie = "Jan";          //przypisanie wartości polu imie dla danego obiektu
    uczl.nazwisko = "Kowalski";
    uczl.wiek = 21;

    cout << "Dane osobowe studenta: " << uczl.imie << " "<<uczl.nazwisko<<" "<<uczl.wiek;
    return 0;
}
```

Istnieją tylko dwie różnice między klasą a strukturą: inne słowo kluczowe podczas deklaracji oraz **domyślny dostęp do składników**. Domyślnie w strukturze wszystkie jej składniki są publiczne, tzn. można się do nich odwołać z poza ciała struktury. W klasach wszystkie składniki domyślnie są prywatne, chyba że programista będzie chciał inaczej.

Klasy są najważniejszym pojęciem w języku C++. Umożliwiają one bowiem ukrywanie, czyli tzw. enkapsulację lub hermetyzację wybranych danych i funkcji tak, aby były one dostępne tylko dla danej klasy.

Główną zaletą hermetyzacji jest ochrona pól obiektów klasy przed dostępem z zewnątrz. Można tak zaprojektować klasę, że dostęp do pól obiektu będzie możliwy tylko przez odpowiednie funkcje.

Do zdefiniowania dostępu do składników klasy używa się 3 słów kluczowych:

- **public** – publiczne składniki mogą być używane zarówno z zawartości klasy, jak i spoza jej zakresu, czyli są dostępne bez ograniczeń; zwykle takimi składnikami są wybrane funkcje składowe, za ich pomocą przeprowadza się z zewnątrz operacje na danych typu **private**;
- **private** – deklarowane składniki są dostępne tylko z wnętrza klasy; w przypadku danych składowych oznacza to, że tylko funkcje będące składnikami klasy mogą te prywatne dane odczytywać lub do nich zapisywać; jeżeli zależy nam na ukryciu informacji, wówczas składnik powinien być zadeklarowany jako **private**;
- **protected** – deklarowane składniki są dostępne tak jak składniki typu **private**, ale dodatkowo są dostępne dla klas wywodzących się od tej klasy.

Uwaga!
Jeśli w definicji klasy nie wystąpi żadna z etykiet, to składniki przez domniemanie mają dostęp **private**.

Aby zastosować operatory dostępu, po prostu poprzedza się nimi odpowiednie składniki:

PRZYKŁAD 9.4

```
class student{
public:                        //składniki publiczne
string imie;
string nazwisko;
int wiek;
int wzrost;
void wyświetl();
private:                  //składniki prywatne
int pesel;
};
```

Rys. 9.3. Składniki publiczne i prywatne w klasie

SPRAWDŹ SWOJĄ WIEDZĘ

1. Co to jest klasa?
2. Wymień składniki klasy.
3. Scharakteryzuj słowa kluczowe używane do zdefiniowania dostępu do składników klasy.

9.2 Klasa, obiekt i funkcje w klasach

ZAGADNIENIA
- Klasa a obiekt
- Funkcje w klasach

Klasa a obiekt

Jeśli mamy zdefiniowaną klasę, możemy stworzyć obiekty tej klasy, tak jak w przypadku typu **int** możemy stworzyć kilka obiektów typu całkowitego:

```
int a, b, c, wiek;
```

Jeśli jest już utworzona klasa **student**, to obiekty tej klasy można zdefiniować w następujący sposób:

```
student osoba1, osoba2, osoba3;
```

Sama definicja klasy nie definiuje żadnych obiektów. Po definicji klasy **student** nie ma jeszcze w pamięci żadnej zmiennej **wiek** czy **nazwisko**.

Uwaga!
Klasa to typ obiektu, a nie sam obiekt.

Odwołanie do składowych klasy wprowadza się za pomocą kropki:

```
nazwa_egemplarza_klasy.nazwa_składowej=wartość;
```

PRZYKŁAD 9.5

```
#include <iostream>
#include <conio.h>
#include <string>
using namespace std;
class student
{
      public:
         string imie;
         string nazwisko;
         int wiek;
};

int main()
{
   student osoba1;
   osoba1.imie="Anna";              //odwołanie się do pola imie dla obiektu osoba1
   osoba1.nazwisko="Nowak";         //odwołanie się do pola nazwisko dla obiektu osoba1
   osoba1.wiek=33;                  //odwołanie się do pola wiek dla obiektu osoba1
   cout<<"Dane studenta: "<<osoba1.imie<<" "<<osoba1.nazwisko<<" "<<osoba1.wiek<<" lata"<<endl;
  return 0;
}
```

Rys. 9.4. Definiowanie obiektu klasy i odwołanie sie do poszczególnych pól obiektu

Efekt działania powyższego programu jest następujący:

```
Dane studenta: Anna Nowak 33 lata
```

Rys. 9.5. Wynik działania programu

Funkcje w klasach

Funkcje zadeklarowane wewnątrz definicji klasy są składnikami tej klasy i nazywa się je **funkcjami składowymi** lub **metodami**. Funkcje te tworzymy na takiej samej zasadzie jak zwykłe funkcje. Są to po prostu narzędzia, za pomocą których wykonujemy operacje na danych składowych klasy. Metodę można zdefiniować albo wewnątrz klasy lub poza nią. W przypadku definicji poza klasą należy zastosować operator zasięgu. Ważne, aby jej deklaracja znajdowała się w ciele klasy.

PRZYKŁAD 9.6

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

class punkt                              //definicja klasy punkt
{
public:
    float x,y;                           //pola klasy

    void wypisz()                        //metoda klasy punkt
    {
        cout<<"Punkt ("<<x<<", "<<y<<")"<<endl;
    }

};
int main()
{
    punkt P;                             //zdefiniowanie egzemplarza klasy punkt
    P.x=5.5;                             //przypisanie polu x wartości 5,5
    P.y=3;                               //przypisanie polu y wartości 3
    P.wypisz();
    return 0;
}
```

Rys. 9.6. Definicja funkcji wewnątrz klasy

SPRAWDŹ SWOJĄ WIEDZĘ

1. Co to jest obiekt klasy?
2. Scharakteryzuj funkcje w klasach.

9.3 Konstruktor i destruktor

ZAGADNIENIA
- **Konstruktor**
- **Cechy konstruktora**
- **Destruktor**
- **Cechy destruktora**

Aby klasa, jako typ definiowany przez użytkownika, przypominała typy wbudowane, utworzono trzy rodzaje funkcji składowych:
a) konstruktor i destruktor;
b) funkcje składowe przeładowujące operatory;
c) operatory konwersji.

Kiedy klasa jest inicjowana, zawsze istnieje konieczność wypełnienia jej odpowiednimi danymi. W tym celu nie trzeba posługiwać się osobnymi metodami, można skorzystać z konstruktora.

Konstruktor to specjalna funkcja składowa, która nazywa się tak samo jak klasa. W ciele tej funkcji można zamieścić instrukcje nadające składnikom obiektu wartości początkowe.

Uwaga!
Konstruktor nie przydziela pamięci obiektowi. On tylko inicjuje pamięć.

Jeśli klasa ma odpowiedni konstruktor, to jest on automatycznie uruchamiany podczas definiowania każdego obiektu tej klasy. Nie można wywołać konstruktora na rzecz już istniejącego obiektu. Nie można pobrać adresu konstruktora. Konstruktor nie jest widoczny w zakresie klasy.

Cechy konstruktora

- Konstruktor może być przeładowany.
- Konstruktor nie ma typu zwracanej wartości (nawet typu **`void`**).
- Konstruktor może być wywołany dla tworzenia obiektów z przydomkami **`const`** oraz **`volatile`**, ale sam nie może być funkcją typu **`const`** i **`volatile`**.
- Konstruktor nie może być typu **`static`**, ponieważ pracuje na niestatycznych składnikach klasy.

Nazwa konstruktora musi być taka sama jak nazwa klasy, a jego definicja – znajdować się w sekcji **`public`**. Istnieje wiele rodzajów konstruktorów. Można je zdefiniować jawnie, co daje pełną kontrolę nad jego wyglądem i zachowaniem. W przypadku braku konstruktora zostanie on niejawnie dodany do klasy automatycznie podczas kompilacji programu.

PRZYKŁAD 9.9

```
#include <iostream>
using namespace std;
class osoba
{
   string imie, nazwisko;
   int wiek;
   public:
    osoba(string="Janek",string="Dzbanek",int=10);       //deklaracja konstruktora
    ~osoba();                                            //deklaracja destruktora
    void dane();
   };
void osoba::dane()
{
   cout<<endl<<imie<<" "<<nazwisko<<" ma "<<wiek<<" lat.";
}

osoba::osoba(string i,string n,int w)                    //definicja kostruktora
{
   imie=i;
   nazwisko=n;
   wiek=w;
}
osoba::~osoba()                                          //definicja destruktora
{
   cout<<endl<<"DESTRUKTOR zostal uruchomiony i na koniec skasowal obiekty";
}
int main()
{
    osoba o1;
    o1.dane();
    return 0;
}
```

SPRAWDŹ SWOJĄ WIEDZĘ

1. Co to jest konstruktor?
2. Wymień cechy konstruktora.
3. Do czego służy destruktor?

9.4 Tablice obiektów klasy

ZAGADNIENIA

- Tworzenie tablic z obiektów
- Agregat
- Przykłady tablic obiektów klasy

W języku C++ można tworzyć tablice z obiektów typów wbudowanych, a także ze wskaźników do tych obiektów:

```
int tablica[10];        // 10-elementowa tablica obiektów typu int;
float tab[5];           // 5-elementowa tablica obiektów typu float;
char *wsk[4];           // 4-elementowa tablica wskaźników do char.
```

Można także tworzyć tablice obiektów danej klasy.

PRZYKŁAD 9.10

```
#include <iostream>

using namespace std;
class student
{
      public:
         string imie;
         string nazwisko;
         int wiek;

void wyswietl();
};

int main()
{
    student osoby[3];          //deklaracja trzyelementowej tablicy obiektów klasy student
    return 0;
}
```

Rys. 9.11. 3-elementowa tablica obiektów

Każdy z elementów tablicy ma trzy pola: **imie**, **nazwisko**, **wiek** oraz funkcję składową **wyswietl()**.

Tablica obiektów jest **agregatem** (czyli skupiskiem danych), gdy:

- nie ma składników danych prywatnych (**private**) lub zastrzeżonych (**protected**);
- nie ma konstruktorów;
- nie ma klas podstawowych;
- nie ma funkcji wirtualnych.

Takie warunki spełnia klasa z poprzedniego przykładu, dlatego wiadomo, że tablica obiektów tej klasy też jest agregatem i może być inicjowana przez listę wartości w nawiasach klamrowych w odpowiedniej kolejności. Jeśli podamy za mało wartości, to reszta zostanie zainicjowana zerami. Inicjujemy ją przez podanie wartości składowych dla kolejnych obiektów.

PRZYKŁAD 9.11

```
#include <iostream>

using namespace std;
class student
{
      public:
         string imie;
         string nazwisko;
         int wiek;

void wyswietl();
};

int main()
{
    student osoby[3]={{"Anna","Nowak",30},{"Joanna","Kowalska",35} ,{"Jan","Nowak",40}};

    for(int i=0;i<3;i++)
    {
        cout<<osoby[i].imie<<" "<<osoby[i].nazwisko<<" ma "<<osoby[i].wiek<<" lat."<<endl;
    }

    return 0;
}
```

Rys. 9.12. Wprowadzanie danych do tablicy obiektów

Tak jak w przypadku zwykłych tablic, wprowadzanie i wyprowadzanie wartości elementów wykonuje się za pomocą pętli **FOR**. Efekt działania powyższego programu jest następujący:

```
Anna Nowak ma 30 lat.
Joanna Kowalska ma 35 lat.
Jan Nowak ma 40 lat.

Process returned 0 (0x0)   execution time : 0.078 s
Press any key to continue.
```

Rys. 9.13. Wyświetlenie wartości tablicy obiektów

Oczywiście wartości przypisane elementom tablicy obiektów (podane przez użytkownika) można wprowadzić z klawiatury.

PRZYKŁAD 9.12

```
#include <iostream>
using namespace std;
class student
{
      public:
         string imie;
         string nazwisko;
         int wiek;

void wyswietl();
};
int main()
{
    int i;
    student osoby[3];
    for ( i=0;i<3;i++)
    {
        cout<<"Podaj dane studenta nr "<<i+1<<endl;
        cout<<"Imie: ";
        cin>>osoby[i].imie;
        cout<<"Nazwisko: ";
        cin>>osoby[i].nazwisko;
        cout<<"Wiek: ";
        cin>>osoby[i].wiek;
        cout<<endl;
    }
    for(i=0;i<3;i++)
    {
        cout<<osoby[i].imie<<" "<<osoby[i].nazwisko<<" ma "<<osoby[i].wiek<<" lat."<<endl;
    }
    return 0;
}
```

Rys. 9.14. Zastosowanie pętli FOR do pracy na tablicy obiektów

Program wczytuje z klawiatury poszczególne elementy tablicy obiektów (pierwsza pętla), a następnie wartości te zostają wypisane na ekranie:

```
Podaj dane studenta nr 1
Imie: Anna
Nazwisko: Nowak
Wiek: 20

Podaj dane studenta nr 2
Imie: Katarzyna
Nazwisko: Kowalska
Wiek: 19

Podaj dane studenta nr 3
Imie: Jan
Nazwisko: Buzek
Wiek: 20

Anna Nowak ma 20 lat.
Katarzyna Kowalska ma 19 lat.
Jan Buzek ma 20 lat.

Process returned 0 (0x0)   execution time : 35.641 s
Press any key to continue.
```

Rys. 9.15. Wyświetlenie danych wprowadzonych przez użytkownika

Wartości wczytane do tablicy obiektów danej klasy mogą być argumentami funkcji spoza klasy.

PRZYKŁAD 9.13

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

class student
{
      public:
         string imie;
         string nazwisko;
         int wiek;

void wyswietl();
};

void srednia(int a,int b,int c)
{
    float sr=(a+b+c)/3;
    cout<<"Srednia wieku studentow wynosi: "<<sr<<" lat.";
}
int main()
{
    int i;
    student osoby[3]={{"Anna","Nowak",21},{"Katarzyna","Kowalska",20},{"Jan","Buzek",19}};


    for(i=0;i<3;i++)
    {
        cout<<osoby[i].imie<<" "<<osoby[i].nazwisko<<" ma "<<osoby[i].wiek<<" lat."<<endl;
    }
    srednia(osoby[0].wiek,osoby[1].wiek,osoby[2].wiek);
    return 0;
}
```

Rys. 9.16. Zastosowanie tablicy obiektów

Funkcja **srednia** jest zdefiniowana ze zwykłymi argumentami typu **int**, ale została wywołana z elementami tablicy obiektów.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Jak zdefiniujemy tablicę obiektów? Podaj przykład.
2. Co to znaczy, że tablica obiektówjest agregatem?

9.5 Dziedziczenie

ZAGADNIENIA

- **Co to jest dziedziczenie?**
- **Dziedziczenie publiczne**
- **Dziedziczenie prywatne**

Dziedziczenie to technika pozwalająca na definiowanie nowej klasy z wykorzystaniem klasy już istniejącej. Klasę dziedziczącą nazywamy **klasą pochodną**, a klasę, po której klasa pochodna dziedziczy – **klasą bazową**. Klasa pochodna pochodzi od bazowej.

W klasie pochodnej można:

- zdefiniować dodatkowe dane składowe;
- zdefiniować dodatkowe funkcje składowe;
- zdefiniować składnik, który istnieje już w klasie podstawowej.

PRZYKŁAD 9.14

```
class student                          //klasa bazowa
{
      public:
         string imie;
         string nazwisko;
         int wiek;

void wyswietl();
};

class dod_inf: public student          //klasa pochodna
{
 public:
     int wzrost;
     void sprawdz();
};
```

Rys. 9.17. Klasa bazowa i klasa pochodna

Klasa **dod_inf** wywodzi się od klasy **student**, czyli klasa **dod_inf** jest klasą pochodną klasy **student**. W definicji klasy **dod_inf** po dwukropku jest umieszczona tzw. lista pochodzenia, czyli informacja, od jakiej klasy wywodzi się klasa pochodna, oraz specyfikacja dostępu, czyli sposób dziedziczenia.

W dziedziczeniu publicznym (najczęściej stosowane) składowe publiczne klasy bazowej są odziedziczone jako publiczne, a składowe chronione (**protected**) – jako chronione.

W dziedziczeniu chronionym składowe publiczne są dziedziczone jako chronione, a składowe chronione – jako chronione.

Dziedziczenie prywatne jest domyślne, jeśli nie jest podany typ dziedziczenia. Składowe publiczne są dziedziczone jako prywatne, a chronione jako prywatne.

We wszystkich sposobach dziedziczenia składowe prywatne klasy bazowej są dziedziczone jako prywatne, ale klasa pochodna nie ma do nich dostępu. Aby klasa pochodna miała dostęp do prywatnych składowych klasy bazowej, musi być zaprzyjaźniona z klasą bazową.

Tabela 9.1. Działanie dziedziczenia

		Sposób dziedziczenia		
		public	protected	private
Widoczność w klasie bazowej	public	public	protected	private
	protected	protected	protected	private
	private	brak	brak	brak

PRZYKŁAD 9.15

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

class figura
{
protected:
    float podstawa, wysokosc;
public:
    void wypisz()
    {
        cout<<"Podstawa figury wynosi: "<<podstawa<<" wysokosc wynosi: "<<wysokosc<<endl;
    }
     void przypisz(float a, float h)
    {
        podstawa = a;
        wysokosc = h;
    }
};

class kwadrat :public figura
{
public:
    float pole()
    {
         return podstawa * wysokosc;
    }
};
```

```
class trojkat :public figura
{
public:
    float pole()
    {
        return (podstawa * wysokosc)/2;
    }
};

int main()
{
    kwadrat figura1;
    trojkat figura2;

    figura1.przypisz(4,4);
    figura2.przypisz(4,4);
    figura1.wypisz();
    figura2.wypisz();


    cout<<"Pole kwadratu wynosi: "<<figura1.pole()<<endl;
    cout<<"Pole trojkata wynosi: "<<figura2.pole();

    return 0;
}
```

Rys. 9.18. Zastosowanie dzidziczenia

SPRAWDŹ SWOJE UMIEJĘTNOŚCI

1. Napisz program zawierający klasę **punkt** z dwoma polami typu rzeczywistego oraz metodą wyświetlającą współrzędne punktu. Zdefiniuj dwa obiekty tej klasy, wczytaj dla nich dane z klawiatury i za pomocą zwykłej funkcji oblicz odległość między punktami.
2. Napisz program zawierający klasę **figura** z trzema polami typu rzeczywistego oraz metodą wyświetlającą długości boków figury, a także metodą obliczającą obwód trójkąta. Zdefiniuj dwa obiekty tej klasy, wczytaj dla nich dane z klawiatury i za pomocą zwykłej funkcji sprawdź, która z figur ma większy obwód.

SPRAWDŹ SWOJĄ WIEDZĘ

1. Na czym polega dziedziczenie?
2. Co to jest klasa bazowa i pochodna?
3. Co oznaczają słowa kluczowe **private**, **protected**, **public**?

9.6 Polimorfizm

ZAGADNIENIA
- **Polimorfizm**
- **Klasa abstrakcyjna**
- **Funkcja wirtualna**
- **Wiązania statyczne i dynamiczne**

Jedną z cech charakterystycznych dla programowania obiektowego jest polimorfizm. Wyraz „polimorfizm", pochodzący z języka greckiego, oznacza „wielopostaciowość".

W języku C++, tak jak w innych językach obiektowych, polimorfizm (wielopostaciowość) oznacza, że dany obiekt może zmieniać swoją postać w zależności od potrzeb w programie. Polimorfizm w tym języku polega na tym, że wywoływane mogą być różne wersje tej samej funkcji. Powstają one w trakcie dziedziczenia, podczas gdy klasy pochodne w specjalny sposób redefiniują pewną funkcję składową.

Polimorfizm wiąże się z zastosowaniem polimorficznego wskaźnika do tzw. abstrakcyjnej klasy podstawowej, czyli klasy niereprezentującej żadnego konkretnego obiektu.

Rozważmy zbiór klas przechowujących informacje o figurach płaskich:
- klasa **kolo**
- klasa **kwadrat**.

Każda z tych klas ma własną funkcję składową **obwod()**, która oblicza obwód wybranej figury. Jeśli chcemy w naszym programie zastosować mechanizm polimorfizmu, wszystkie te klasy muszą być klasami pochodnymi klasy o nazwie **figura**, która ma tzw. wirtualną metodę **obwod()**:

```
class figura
{
public:
   virtual void obwod();
};
class kolo :public figura
{
void obwod ();
};

class kwadrat :public figura
{
void obwod ();
};
```

Można teraz zdefiniować **polimorficzny wskaźnik** typu figura i ustawić go na dowolny obiekt klasy pochodnej:

```
kolo f1;          //zdefiniowanie obiektu klasy kolo
kwadrat f2;             //zdefiniowanie obiektu klasy kwadrat
figura *wsk=&f1;//ustawienie wskaźnika na obiekt klasy kolo
wsk-> obwod();          //wywołanie metody obwod ()
```

Wywołanie funkcji składowej **obwod()** może działać różnie w zależności od typu obiektu, na który wskazuje polimorficzny wskaźnik.

Uwaga!

Nie można definiować obiektów klasy figura. Jest ona bowiem tzw. **abstrakcyjną klasą** podstawową, która jedynie służy do utworzenia polimorficznego wskaźnika.

Klasa figura posiada **wirtualną funkcję składową**. Jest to funkcja, która nie będzie nigdy wykonana, a tylko zostanie przesłonięta przez funkcje **obwod()** znajdujące się w poszczególnych klasach pochodnych. Słowo ***virtual*** jest informacją dla kompilatora, że metoda ta będzie zastąpiona metodą o tej samej nazwie pochodzącej od potomka.

Metoda wirtualna jest czysta, gdy przypiszemy jej wartość zerową:

```
virtual void obwod()=0;
```

Nie ma bowiem sensu pisać kodu tej funkcji, ponieważ i tak nie zostanie ona wykonana.

Gdy typ obiektu, dla którego wywoływana jest funkcja, jest znany podczas kompilacji, czyli:

```
kolo f1;
f1. obwod();
```

to jest to tzw. **wiązanie statyczne**. Wiadomo bowiem, że **f1** jest obiektem klasy **kolo**.

Natomiast użycie wskaźnika, czyli zastosowanie zapisu:

```
figura *wsk;
wsk=&f1;
wsk-> obwod();
```

to tzw. **wiązanie dynamiczne** (inaczej późne wiązanie). Wskaźnik **wsk** można bowiem przestawić na obiekt dowolnej klasy pochodnej.

Polimorfizm to mechanizm pozwalający zaprojektować program, który będzie łatwy do rozbudowania (rozszerzenia). Oczywiście polimorfizm jest potrzebny w dużych projektach, tam gdzie cecha rozszerzalności programu jest bardzo dużą zaletą.

Zastosowanie mechanizmu polimorfizmu ilustruje następujący kod źródłowy (Przykład 9.16):

PRZYKŁAD 9.16

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

class figura                    // abstrakcyjna klasa podstawowa
{
public:
   virtual void obwod()=0;       //czysta funkcja wirtualna
};

class kolo :public figura          //klasa pochodna o nazwie kolo
{
float r;
public:
   kolo(float a)                //konstruktor klasy kolo
   {
      r=a;
   }
   void obwod()
   {
      cout<<"Obwod kola o promieniu "<<r<<" wynosi: "<<2*3.14*r<<endl;
   }
};

class kwadrat :public figura       //klasa pochodna o nazwie kwadrat
{
float a;
public:
   kwadrat(float x)                // konstruktor klasy kwadrat
   {
      a=x;
   }
   virtual void obwod()
   {
      cout<<"Obwod kwadratu o boku: "<<a<<" wynosi: "<<4*a<<endl;
   }
};

int main()
{
    kolo figura1(2);                //zdefiniowanie obiektu klasy kolo o promieniu 2
    kwadrat figura2(2);             //zdefiniowanie obiektu klasy kwadrat o boku 2

    figura *wsk;
    wsk = &figura1;                 //ustawienie wskaźnika na obiekt klasy kolo
    wsk -> obwod();                  //wywolanie metody obwod dla obiektu figura1

    wsk = &figura2;                 //ustawienie wskaźnika na obiekt klasy kwadrat
    wsk -> obwod();                  //wywolanie metody obwod dla obiektu figura2

    return 0;
}
```

Rys. 9.19. Zastosowania mechanizmu polimorfizmu

SPRAWDŹ SWOJĄ WIEDZĘ:

1. Co to jest klasa abstrakcyjna?
2. Opisz funkcję wirtualną oraz jej zastosowanie.
3. Jaka jest różnica między wiązaniem statycznym a dynamicznym?
4. Na czym polega mechanizm polimorfizmu?

10. Projektowanie i dokumentowanie programów

10 Projektowanie i dokumentowanie programów

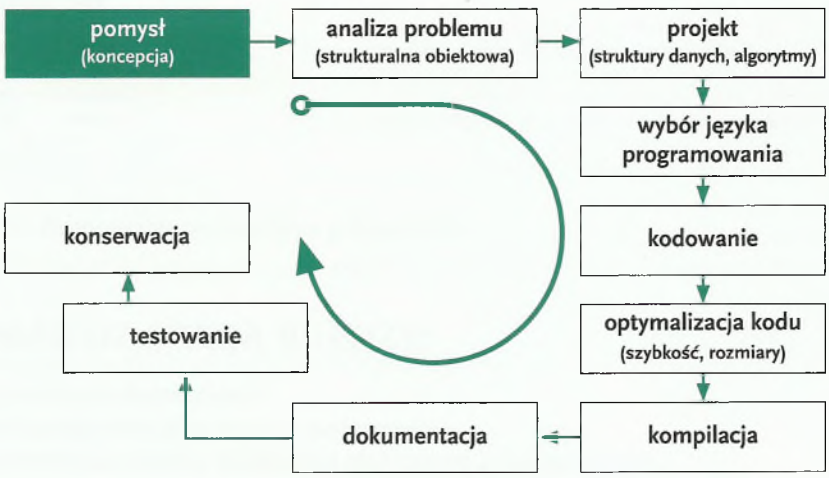
ZAGADNIENIA

- Etapy projektowania
- Planowanie
- Implementacja
- Kompilacja
- Konsolidacja
- Testowanie
- Dokumentowanie programów
- Przykład realizacji prostej aplikacji

Etapy projektowania

Pracę nad programem można podzielić na następujące etapy:

1. planowanie – zdefiniowanie problemu, który chcemy rozwiązać, oraz jego dokładna analiza i wybór metody jego rozwiązania;
2. implementacja (kodowanie) – zapisanie algorytmu w postaci kodu źródłowego;
3. kompilacja – zamiana kodu źródłowego na binarny;
4. konsolidacja – łączenie plików obiektowych i biblioteki w program wykonywalny;
5. testowanie – wykrywanie błędów;
6. wdrażanie i optymalizacja – ulepszenie programu;
7. użytkowanie i ewaluacja.



Rys. 10.1. Etapy tworzenia programu

Planowanie

Pierwszym etapem tworzenia programu komputerowego jest określenie problemu, który należy rozwiązać. Formułujemy problem, który jest rozwiązywany za pomocą algorytmiki. Program komputerowy powinien być funkcjonalny, a zadanie stawiane programiście powinno dać się wyrazić za pomocą algorytmu i musi być możliwe do rozwiązania.

Implementacja

Kodowanie jest bardzo ważnym etapem, ponieważ wymaga jego dokładnego udokumentowania. Zapis powinien być czytelny. Ważne jest właściwe zorganizowanie kodu źródłowego. Należy stosować odpowiednie komentarze, tak aby osoba przejmująca kodowanie potrafiła szybko zorientować się w nazwach zmiennych oraz funkcji. Ponadto wcięcia ułatwiają znajdowanie instrukcji, co przyspiesza wyszukiwanie błędów oraz modyfikację programu.

Kompilacja

Podczas pisania programu konieczne jest przetestowanie fragmentów kodu pod względem poprawności syntaktycznej, obecności ewentualnych błędów i zgodności działania z zamierzeniem programisty.

Konsolidacja

Konsolidacja, czyli linkowanie (łączenie), polega na łączeniu poszczególnych modułów programu w celu wygenerowania programu. Istnieją dwa rodzaje konsolidacji.

- **Konsolidacja statyczna** – najczęściej przeprowadzana podczas tworzenia programu i będąca procesem dołączania modułów bibliotecznych do wykonywanego programu oraz łączenia wszystkich modułów składowych samego programu.
- **Konsolidacja dynamiczna** – zachodzi podczas ładowania programu do pamięci, przy współpracy systemu operacyjnego; umożliwia lepsze wykorzystanie pamięci oraz ułatwia aktualizowanie bibliotek.

Testowanie

Testowanie programu odgrywa ogromną rolę zarówno w procesie tworzenia aplikacji, jak i podczas użytkowania. Testowanie ma na celu wykrycie błędów, tzn. takich jego elementów, które mogą prowadzić do niewłaściwego działania programu.

Optymalizacja

Optymalizacja programu to czynności, których celem jest ulepszenie i poprawa wydajności programu komputerowego. Optymalizację działania programu uzyskujemy np. poprzez zmniejszenie ilości przydzielanej pamięci, a także przez zwiększenie tempa działania programu czy usunięcie zbędnych linii kodu.

Dokumentowanie tworzonych aplikacji

Wszystkie działania twórcy programu mające na celu opisanie rozwiązań programistycznych oraz wskazań dla przyszłego użytkownika sprowadzają się do tworzenia:

1. dokumentacji technicznej programu,
2. dokumentacji obsługi programu dla użytkownika.

Dokumentacja techniczna programu powinna zawierać oprócz komentarzy wewnątrz kodu źródłowego również opis projektu, którego efektem końcowym jest program. Dokumentacja taka zazwyczaj jest znormalizowana (np. ISO 9000) i dotyczy wszystkich etapów tworzenia aplikacji.

Na etapie realizacji projektu mającego na celu wytworzenie określonego programu dokumentację techniczną tworzyć mogą:

- opisy wymagań zleceniodawcy lub zapytania przetargowe;
- specyfikacje techniczne (np. zastosowany język programowania), architektura aplikacji,
- modele, schematy (np. diagramy UML), procedury czy algorytmy;
- raporty, umowy, maile czy inne formy komunikacji między zleceniodawcą a autorami aplikacji;
- listingi kodu źródłowego wraz z komentarzami;
- opis procesu testowania (np. raporty z testów).

Prawidłowo sporządzona dokumentacja techniczna jest podstawowym źródłem informacji dla programistów w przypadku naprawy lub modyfikacji gotowego programu.

W skład dokumentacji obsługi programu wchodzą:

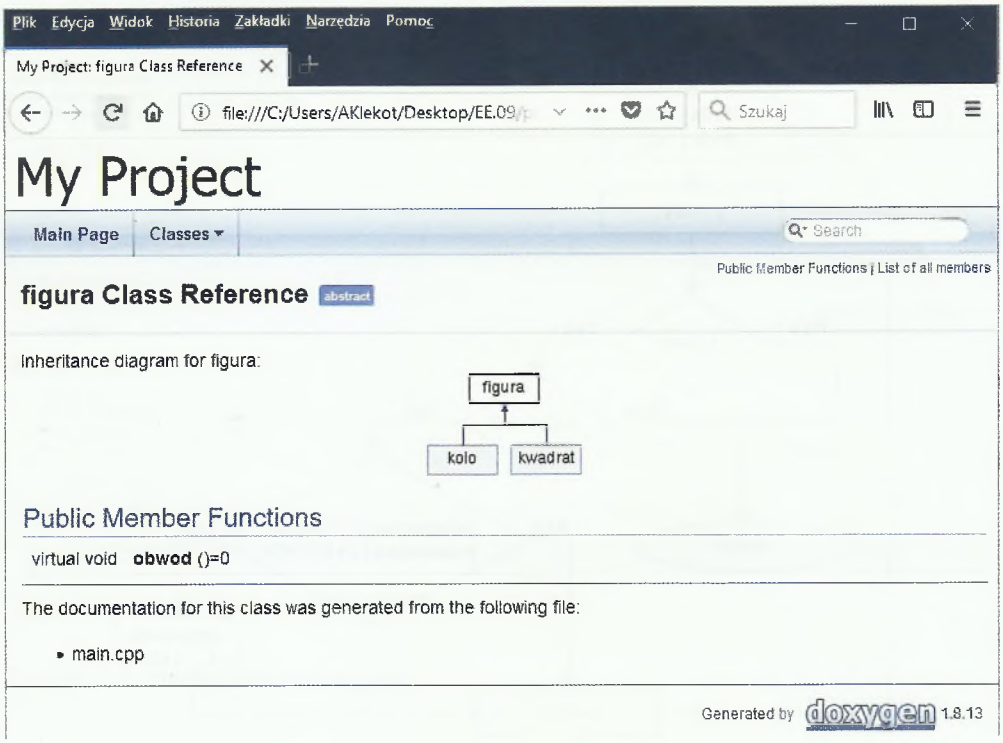
- ogólny funkcjonalny opis systemu,
- wymagania sprzętowe,
- instrukcja obsługi programu,
- przewodnik instalacji i opis spotykanych problemów,
- opis dokumentujący daną wersję programu.

Dokumentacja obsługi często dostępna jest w formie elektronicznej. Powinna mieć szczegółowy spis treści oraz indeks pojęć.

Dużym ułatwieniem w tworzeniu dokumentacji technicznej programów jest wykorzystanie specjalnie do tego przeznaczonych aplikacji np. Doxygen.

Doxygen jest aplikacją dokumentowania oprogramowania pisanego w takich językach, jak C++, C, Java, Objective-C, Python, IDL, Fortran, VHDL, PHP, C# oraz D. Aplikacja może generować dokumentację w HTML, RTF, PDF oraz źródła dla edytora LATEX.

Dokumentacja może być tworzona bezpośrednio ze źródeł, jak też plików stowarzyszonych z nimi. Zastosowanie oprogramowania do dokumentacji typu Doxygen ma sens przy bardzo złożonych kodach źródłowych. Przy prostych aplikacjach główną metodą pozostaje komentowanie określonych części kodu.

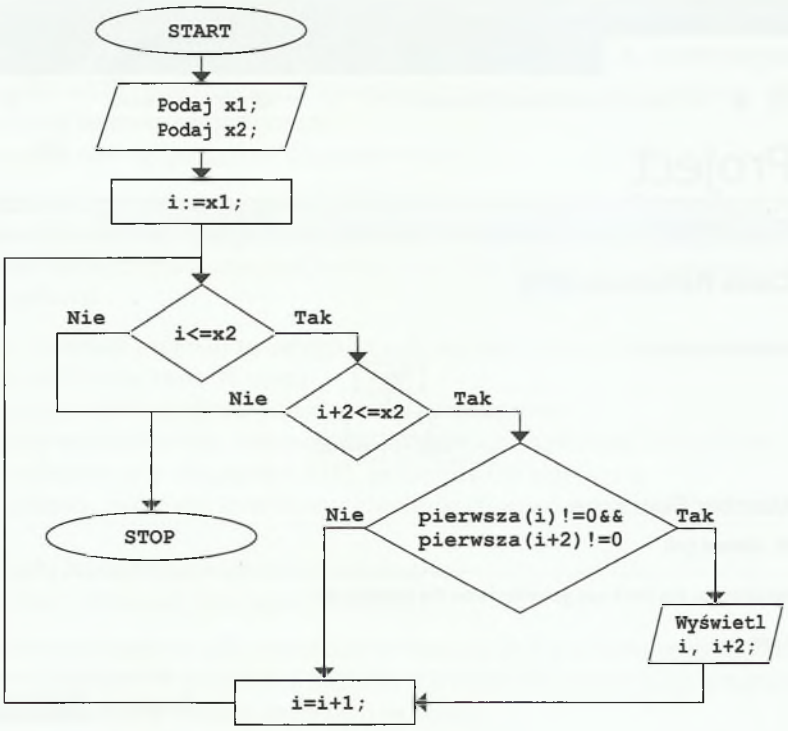


Rys. 10.2. Przykład dokumentacji tworzonej w aplikacji Doxygen

Przykład realizacji prostej aplikacji

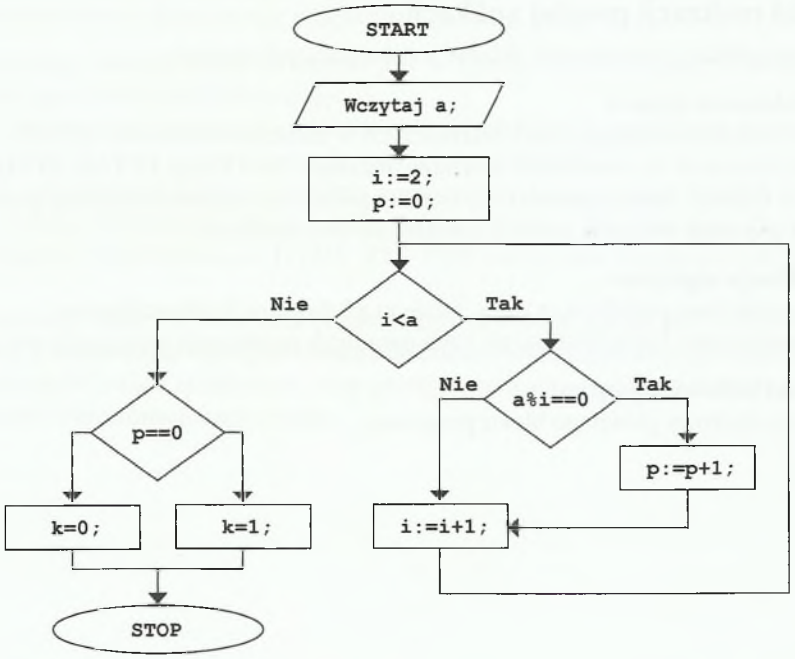
Tworzenie aplikacji powinno się składać z następujących etapów:

1. **Sformułowanie zadania**
   Napisz program szukający liczb bliźniaczych w zadanym przedziale <x1, x2>.
   Liczby bliźniacze to dwie liczby pierwsze, różniące się o 2 (np. 3 i 5 czy 29 i 31).
   Utwórz funkcje, która sprawdzi czy liczba z podanego zakresu jest liczba pierwsza.
   Liczba pierwsza to liczba, która ma dokładnie dwa dzielniki.
2. **Specyfikacja algorytmu**
   Dane wejściowe: przedział <x1,x2>, gdzie x1,x2 dowolne liczby całkowite
   Dane wyjściowe: liczby bliźniacze, czyli para liczb pierwszych różniących się o 2
3. **Schemat blokowy algorytmu**
   Schemat blokowy głównego bloku programu.



Rys. 10.3. Schemat blokowy algorytmu głównego bloku programu

Schemat blokowy funkcji **`pierwsza(int a)`**, sprawdzającej, czy dana liczba jest liczbą pierwszą.



Rys. 10.4. Schemat blokowy algorytmu funkcji sprawdzającej, czy dana liczba jest liczbą pierwszą

4. Implementacja kodu na podstawie napisanych schematów blokowych

```
#include <iostream>
using namespace std;
int pierwsza(int a)
{
    int i,k,p=0;
    if(a<2) k=0;
    else
    {
    for(i=2; i<a; i++)          //sprawdzenie czy dana liczba a jest podzielna przez kolejne liczby naturalne, bez 1 i jej samej
    {
        if(a%i==0) p++;
    }
    if(p==0) k=1;
    else  k=0;
    }
    return k;
}
int main(void)
{
    int x1,x2;
    cout<<"Podaj x1= "; cin>>x1;
    cout<<"Podaj x2= "; cin>>x2;
    for(int i=x1;i<=x2;i++)
    {
        if(i+2<=x2)
        {
            if(pierwsza(i)!=0 && pierwsza(i+2)!=0) //wywolanie funkcji pierwsza
            {
            cout<<i<<", "<<i+2<<endl;
            }
        }
}
 return 0;
}
```

Rys. 10.5. Kod źródłowy gotowej aplikacji

WYKAZ PODSTAWOWYCH POJĘĆ W JĘZYKACH POLSKIM, ANGIELSKIM I NIEMIECKIM

JĘZYK POLSKI	JĘZYK ANGIELSKI	JĘZYK NIEMIECKI
agregat	aggregate	das Aggregat
algorytm	algorithm	der Algorithmus
algorytm rekurencyjny	recursive algorithm	der rekursive Algorithmus
argument	argument	der Parameter/das Argument
argument domniemany	Implied argument	der Default-Parameter
destruktor	destroyer	der Destruktor
drzewo algorytmiczne	algorithm tree	der Baum (die Datenstruktur)
dziedziczenie	heirdom	die Vererbung
ewaluacja	evaluation	die Evaluierung Evaluierung
funkcja	function	die Funktion
funkcja składowa	component function	die Komponentenfunktion
hermetyzacja danych	encapsulation of data	die Datenkapselung
implementacja	implementation	die Implementierung
inicjalizacja	initialization	die Initialisierung
instrukcja warunkowa	conditional statement	die Bedingte Anweisung
klasa bazowa	base class	die Basisklasse
klasa pochodna	erived class	die abgeleitete Klasse
kompilacja	compilation	die Kompilierung
kompilator	compiler	der Compiler
konsolidacja	consolidation	die Konsolidierung
konsolidator	linker	der Linker/ der Binder
konstruktor	constructor	der Konstruktor
konstruktor domyślny	default constructor	der parameterlose Konstruktor
łańcuch znaków	string of characters	die Zeichenkette
metoda	method	die Methode
obiekt	object	das Objekt
operator	operator	der Operator
operator arytmetyczny	arithmetic operator	der arithmetische Operator

JĘZYK POLSKI	JĘZYK ANGIELSKI	JĘZYK NIEMIECKI
operator binarny	binary operator	der binäre Operator
operator dekrementacji	decrement operator	der Dekrement-Operator
operator inkrementacji	increment operator	der Inkrement-Operator
operator logiczny	logical operator	der logische Operator
operator porównania	comparison operator	der Vergleichsoperator
operator przypisania	assignment operator	der Zuweisungsoperator
operator zasięgu	range operator	der Scope (Resolution) Operator
optymalizacja	optimization	die Optimierung
pętla	loop	die Schleife
plik	file	die Datei
polimorfizm	polymorphism	die Polymorphie
program	program	das Programm
programowanie	programming	die Programmierung
prototyp	prototype	der Prototyp
przeładowanie funkcji	reload function	die Funktion Überlastung
pseudokod	pseudocode	der Pseudocode
referencja	reference	die Referenz
selektor	selector	der Selektor
składowe klasy	class members	die Komponente der Klasse
struktura	structure	der Verbund
tablica	array	das Feld
tablica jednowymiarowa	one-dimensional array	das Eindimensional-Feld
tablica obiektów	array of objects	das Array von Objekten
tablica wielowymiarowa	multidimensional array	das Mehrdimensional-Feld
testowanie	testing	der Softwaretest
wskaźnik	indicator	der Zeiger; der Indikator
złożoność obliczeniowa	computational complexity	die Komplexitätstheorie
zmienna	variable	die Variable
zmienna globalna	global variable	die globale Variable
zmienna lokalna	local variable	die lokale Variable

LITERATURA

Publikacje książkowe:

1. J. Grębosz, *Symfonia C++. Programowania w języku C++ orientowane obiektowo*, Oficyna Kallimach, Warszawa 1993.
2. B. Stroustrup, *Język C++,* WNT, Warszawa 2008.
3. A. Stasiewicz, C++. *Ćwiczenia zaawansowane*, Helion, Gliwice 2005.

Netografia:

http://www.stack.nl/~dmitri/doxygen

REFORMA 2017

Programowanie i tworzenie stron internetowych oraz baz danych i administrowanie nimi

Część 1

Kwalifikacja EE.09

Podręcznik opracowany **zgodnie z nową podstawą programową** kształcenia w zawodzie **technik informatyk**.

Podręcznik do nauki zawodu technik informatyk realizuje treści z zakresu **kwalifikacji EE.09 Programowanie, tworzenie i administrowanie stronami internetowymi i bazami danych**.

W publikacji zamieszczono wiadomości związane z algorytmiką, językiem C++, tablicami jednowymiarowymi i wielowymiarowymi, operacjami na łańcuchach znaków, operacjami na plikach, wskaźnikami oraz klasami C++. Autorzy wsparli wiadomości teoretyczne licznymi przykładami, ćwiczeniami, bogatym materiałem ilustracyjnym oraz schematami i zrzutami ekranowymi.

Kwalifikacje w ramach zawodu **technik informatyk**

Kwalifikacja **EE.08**	Kwalifikacja **EE.09**
Montaż i eksploatacja systemów komputerowych, urządzeń peryferyjnych i sieci	Programowanie, tworzenie i administrowanie stronami internetowymi i bazami danych

WSiP WYDAWNICTWA SZKOLNE i PEDAGOGICZNE

wsip.pl
sklep.wsip.pl
infolinia: 801 220 555

ISBN 978-83-02-17361-5