# Spis treści


1. Wstęp . . . . . 7

1.1. Zaproszenie . . . . . 7
1.2. Pojęcia podstawowe . . . . . 7
1.3. Budowa komputera . . . . . 10
1.3.1. Procesor . . . . . 10
1.3.2. Pamięć operacyjna . . . . . 11
1.3.3. Urządzenia zewnętrzne . . . . . 12
1.4. Reprezentacja danych w pamięci komputera . . . . . 13
1.4.1. Liczby . . . . . 13
1.4.2. Teksty . . . . . 16

2. Programowanie . . . . . 19

2.1 Zmienne i struktury danych . . . . . 19
2.2. Zapisywanie algorytmów . . . . . 22
2.2.1. Schematy blokowe . . . . . 22
2.2.2. Języki programowania . . . . . 25
2.2.3. Poprawność programów . . . . . 26
2.3. Podstawowe składniki programu . . . . . 27
2.3.1. Deklarowanie danych i struktur danych . . . . . 27
2.3.2. Instrukcje . . . . . 27
2.3.3. Struktury sterowania . . . . . 28
2.3.4. Podprogramy . . . . . 34
2.5. Przygotowanie i wykonanie programu . . . . . 35
2.5.1. Edycja tekstu . . . . . 36
2.5.2. Kompilacja . . . . . 36
2.5.3. Wykonanie programu . . . . . 38
2.5.4. Interpretacja programu . . . . . 39
2.6. Język programowania C . . . . . 40

3. Dane . . . . . 43

3.1. Alfabet . . . . . 43
3.2. Identyfikatory . . . . . 43
3.3. Słowa kluczowe . . . . . 44
3.4. Komentarze . . . . . 44
3.5. Typy danych liczbowych . . . . . 44
3.6. Liczby . . . . . 45
3.6.1. Liczby całkowite . . . . . 45
3.6.2. Liczby zmiennopozycyjne . . . . . 46
3.6.3. Znaki alfabetu . . . . . 46
3.7. Łańcuchy znaków . . . . . 47
3.8. Zmienne . . . . . 48
3.9. Stałe . . . . . 49
3.10. Operator przypisania . . . . . 49
3.11. Wskaźniki . . . . . 49
3.12. Referencje . . . . . 52
3.13. Typy wyliczeniowe . . . . . 53
3.14. Typ logiczny . . . . . 54
3.15. Nazywanie typów danych . . . . . 54

4. Wyrażenia . . . . . 57

4.1. Definicja wyrażenia . . . . . 57
4.2. Konwersja wartości . . . . . 57
4.3. Operatory arytmetyczne . . . . . 58
4.4. Operatory zwiększania i zmniejszania . . . . . 59
4.5. Operatory relacyjne . . . . . 60
4.6. Operatory logiczne . . . . . 61
4.7. Operatory bitowe . . . . . 61
4.8. Złożone operatory przypisania . . . . . 63
4.9. Operator warunkowy . . . . . 63
4.10. Operator rozmiaru . . . . . 64
4.11. Operator wyliczenia . . . . . 64
4.12. Priorytety operatorów . . . . . 64

5. Instrukcje . . . . . 67

5.1. Rodzaje instrukcji . . . . . 67
5.2. Instrukcja prosta . . . . . 67
5.3. Instrukcja złożona . . . . . 68
5.4. Instrukcja warunkowa . . . . . 68
5.5. Instrukcja wyboru . . . . . 70
5.6. Instrukcje pętli . . . . . 72

6. Struktury danych . . . . . 77

6.1. Związki pomiędzy danymi . . . . . 77
6.2. Tablice . . . . . 77
6.2.1. Tablice jednowymiarowe . . . . . 77
6.2.2. Tablice wielowymiarowe . . . . . 80
6.3. Struktury . . . . . 82
6.4. Unie . . . . . 86

7. Funkcje . . . . . 89

7.1. Program główny i podprogramy . . . . . 89
7.2. Deklaracja funkcji . . . . . 89
7.3. Definicja funkcji . . . . . 90
7.4. Wywołanie funkcji . . . . . 91
7.4.1. Przekazywanie argumentów funkcji . . . . . 92
7.4.2. Tablice jako argumenty funkcji . . . . . 95
7.4.3. Wartości domyślne argumentów funkcji . . . . . 97
7.4.4. Funkcje rekurencyjne . . . . . 98
7.4.5. Podprogramy wstawiane . . . . . 98
7.5. Wskaźniki funkcji . . . . . 99
7.5.1. Tablice wskaźników funkcji . . . . . 100
7.5.2 Wskaźnik funkcji jako argument funkcji . . . . . 101
7.5.3. Wskaźnik funkcji jako wynik funkcji . . . . . 102
7.6. Przeciążanie identyfikatorów funkcji . . . . . 103
7.6.1. Wzorce funkcji . . . . . 106
7.7. Funkcje biblioteczne . . . . . 107

8. Struktura programu . . . . . 109

8.1. Części programu . . . . . 109
8.2. Preprocesor . . . . . 109

8.2.1. Dołączanie plików bibliotecznych . . . . . 109
8.2.2. Zastępowanie tekstów . . . . . 110
8.2.3. Kompilacja warunkowa . . . . . 111
8.3. Funkcja main . . . . . 112
8.4. Deklaracje identyfikatorów . . . . . 113
8.4.1. Zakres deklaracji . . . . . 113
8.4.2. Zasięg deklaracji . . . . . 114
8.4.3. Przesłanianie identyfikatorów . . . . . 115
8.4.4. Dostęp do obiektów globalnych . . . . . 116
8.5. Lokalizacja zmiennych . . . . . 117
8.6. Dynamiczny przydział pamięci . . . . . 118

9. Wprowadzanie i wyprowadzanie danych . . . . . 121

9.1. Klawiatura . . . . . 121
9.1.1. getch . . . . . 122
9.1.2. getche . . . . . 123
9.1.3. scanf . . . . . 123
9.1.4. gets . . . . . 126
9.2. Monitor ekranowy . . . . . 127
9.2.1. putchar . . . . . 127
9.2.2. puts . . . . . 127
9.2.3. printf . . . . . 128
9.3. Pliki dyskowe . . . . . 131
9.3.1. fopen . . . . . 131
9.3.2. fclose . . . . . 132
9.3.3. fseek . . . . . 133
9.3.4. feof . . . . . 133
9.3.5. fgetc . . . . . 134
9.3.6. fputc . . . . . 134
9.3.7. fscanf . . . . . 135
9.3.8. fprintf . . . . . 136
9.3.9. fread . . . . . 136
9.3.10. fwrite . . . . . 137

DODATKI . . . . . 139

Dodatek A. Kod ASCII . . . . . 139
Dodatek B. Numery techniczne klawiszy . . . . . 141
Dodatek C. Słowa kluczowe . . . . . 144
Dodatek D. Priorytety i łączność operatorów . . . . . 145

# 1. Wstęp

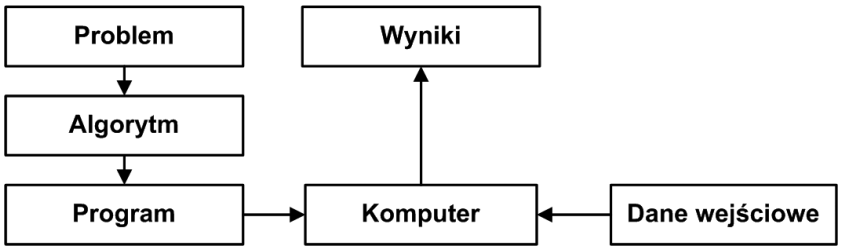
## 1.1. Zaproszenie

Podręcznik niniejszy przeznaczony jest dla czytelników, którzy chcieliby nauczyć się programowania, czyli sposobu tworzenia programów wykonywanych przez komputery. Początkowe rozdziały przedstawiają pojęcia podstawowe dotyczące komputerów i programowania, dalsza część zawiera opis popularnego języka programowania C. Podręcznik ten przeznaczony jest dla szerokiego kręgu czytelników – korzystanie z niego nie wymaga żadnej wstępnej wiedzy specjalistycznej.

## 1.2. Pojęcia podstawowe

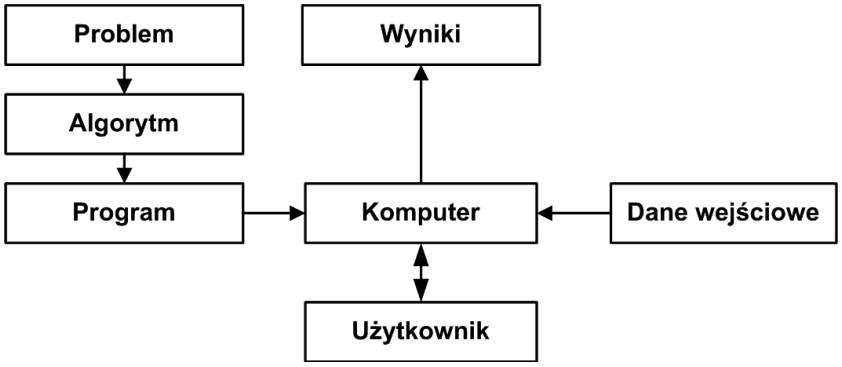
*Komputer* to urządzenie do automatycznego przetwarzania informacji, którego działanie polega na wykonywaniu *programów*. Ta krótka definicja podkreśla dwa istotne fakty. Po pierwsze, komputery służą do przetwarzania informacji, czyli do przetwarzania danych, bowiem *dane* to zapisane w sposób trwały informacje. Proces ten polega na wyznaczaniu nowych danych (wyników, rezultatów), na podstawie danych uprzednio dostępnych (danych wejściowych, początkowych). Po drugie, przebieg procesu przetwarzania danych jest zadawany za pomocą programu, czyli precyzyjnego i jednoznacznego wykazu działań, które komputer ma automatycznie wykonać. Najbardziej nowoczesny komputer jest więc bezużyteczny, o ile nie jest dla niego dostępny żaden program. Z drugiej strony działanie wszystkich komputerów sprowadza się do jednej, zawsze takiej samej czynności – do wykonywania programów.

Abstrakcyjny sposób rozwiązania dowolnego problemu nazywany jest *algorytmem*. Określenie *abstrakcyjny* oznacza w tym przypadku sposób wymyślony przez pewną osobę (lub grupę osób) i pozostający na razie wyłącznie w jej umyśle (w ich umysłach). Algorytm może zostać zapisany, postać tego zapisu zależy najczęściej od rodzaju rozwiązywanego problemu. Na przykład algorytm ugotowania grochówki można zapisać w języku naturalnym, jako przepis kulinarny. Komputery służą, jak już powiedziano, do przetwarzania danych, stąd program wykonywany przez komputer jest zapisem algorytmu rozwiązującego pewien problem przetwarzania danych.



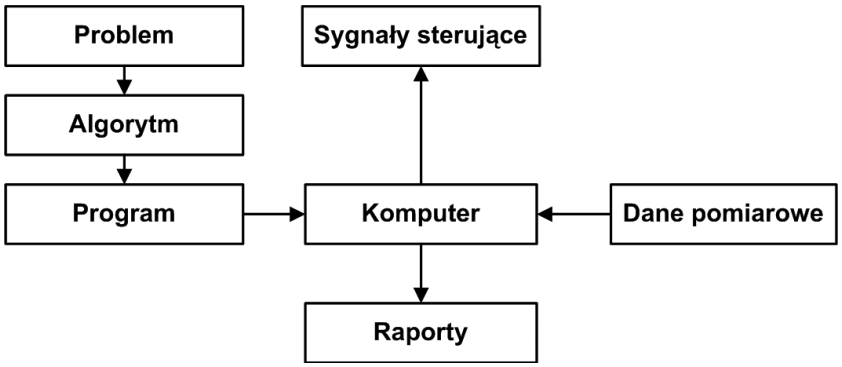

Rys.1.1. Podstawowy schemat wykorzystania komputera

Algorytm, będący rozwiązaniem pewnego problemu, został zapisany jako program (rys. 1.1). Program ten jest wykonywany przez komputer, podczas realizacji programu komputer korzysta z danych wejściowych. Efektem pracy komputera są wyniki przekazywane użytkownikowi (np. wyświetlane na ekranie monitora lub drukowane za pomocą drukarki). Taki schemat użytkowania komputera był przede wszystkim stosowany w początkowym okresie rozwoju informatyki. Obecnie bardzo często użytkownik bierze aktywny udział w wykonaniu programu, dostarczając dodatkowych danych i decydując o kolejnych etapach procesu przetwarzania (rys. 1.2). Typowym przykładem takiego interakcyjnego sposobu wykorzystania komputera są programy edycyjne, umożliwiające przygotowanie, poprawianie, przechowywanie i drukowanie różnych dokumentów.




Rys. 1.2. Interakcyjna współpraca z komputerem

Komputery mogą być również wykorzystywane do sterowania różnymi procesami. Zwane w tym przypadku *mikrokontrolerami* lub *mikroprocesorami* sterującymi nadzorują działanie bardzo wielu popularnych urządzeń, takich jak na przykład telefony komórkowe, telewizory, samochody (np. sterowanie zapłonem, blokada antypoślizgowa ABS), pralki czy urządzenia klimatyzacyjne. Bardzo szeroko komputery są również wykorzystywane do sterowania procesami przemysłowymi. W takich zastosowaniach schemat wykorzystania komputera jest nieco inny (rys. 1.3).




Rys. 1.3. Wykorzystanie komputera do sterowania

Program, będący zapisem pewnego algorytmu sterowania, jest wykonywany przez komputer, który na bieżąco otrzymuje dane pomiarowe. Wynikiem pracy programu są sygnały sterujące i ewentualnie raporty, gromadzone w pamięci trwałej komputera. Na przykład dla wspomnianego układu antypoślizgowego ABS samochodu, dane pomiarowe to ciąg odczytywanych co stały okres czasu wartości prędkości obrotowej koła – wykrycie przez program zbyt gwałtownego zmniejszenia prędkości obrotowej powoduje wygenerowanie sygnału sterującego, zmniejszającego nacisk szczęk hamulcowych, związanych z danym kołem.

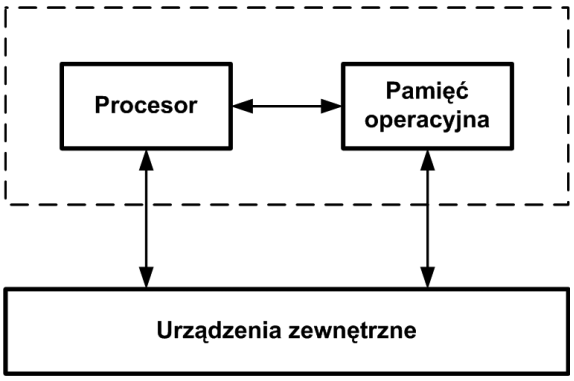
W chwili obecnej komputery to urządzenia elektroniczne (pomijając nieliczne konstrukcje eksperymentalne). Oznacza to, że dane zapisywane są za pomocą odpowiednich stanów układów elektromagnetycznych, wykorzystując ich pojemność, ładunek elektryczny czy namagnesowanie. Przetwarzanie tak zapisanych danych możliwe jest dzięki wykorzystaniu zjawisk elektromagnetycznych – przepływu prądu elektrycznego czy zmiany charakteru namagnesowania. Taki sposób odwzorowania informacji preferuje wykorzystanie do zapisu wartości liczbowych *systemu binarnego (dwójkowego)*, w którym występują tylko dwie cyfry: 0 i 1, zwane *bitami*. Cyfra 0 może być reprezentowana przez brak przepływu prądu lub przez namagnesowanie typu N, cyfrę 1 może reprezentować wystąpienie przepływu prądu lub namagnesowanie typu S. Komputery przechowują więc i przetwarzają wyłącznie ciągi cyfr binarnych (*ciągi zerojedynkowe*, np. 10010101). Oczywiście dla wygody użytkownika dane wejściowe i wyniki pracy programów zapisywane są z użyciem systemu dziesiętnego.

Współpraca użytkownika z komputerem odbywa się za pośrednictwem wyspecjalizowanego programu zwanego *systemem operacyjnym.* System ten zarządza wszystkimi zasobami komputera oraz akceptuje i realizuje polecenia wydawane przez użytkownika. Polecenia te mogą dotyczyć uruchamiania wskazanych programów, zarządzania plikami dyskowymi czy też wykonywania różnych operacji administracyjnych. Początkowo systemy operacyjne udostępniały jedynie interfejs tekstowy – poszczególne polecenia miały postać komend z odpowiednimi parametrami (np. *dir*, *cd E:*, *ipconfig*). Aby uruchomić dany program należało wpisać jako pole-

cenie jego nazwę i podać ewentualne parametry (np. *KopiaPliku DanePrywatne.txt*). Obecnie większość systemów operacyjnych oferuje użytkownikowi interfejs graficzny – poszczególne programy uruchamia się wybierając myszką reprezentujące je znaki graficzne (ikony). Niemniej nadal możliwe jest tekstowe sterowanie pracą komputera za pomocą *okna konsoli* (tzw. *wiersz poleceń*). Programy przykładowe zamieszczone w niniejszym podręczniku są przeznaczone do uruchamiania właśnie w oknie konsoli.

## 1.3. Budowa komputera

Podstawowe elementy komputera przedstawione zostały na rysunku 1.4. Centralną częścią każdego komputera jest *procesor* – układ elektroniczny wykonujący programy. Programy te oraz wymagane przez nie dane, pobierane są z *pamięci operacyjnej*. Wprowadzanie i wyprowadzanie danych, ich przechowywanie czy przesyłanie umożliwiają *urządzenia zewnętrzne*.



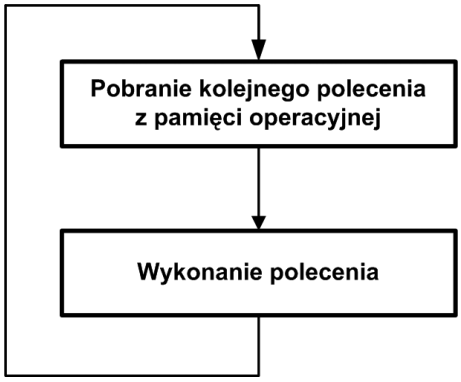

Rys. 1.4. Budowa komputera

### 1.3.1. Procesor

Zadaniem *procesora* jest wykonanie programu. Każdy typ procesora posiada swój charakterystyczny i niepowtarzalny zestaw poleceń (rozkazów), które potrafi wykonywać. Aby program mógł być wykonany przez procesor musi więc mieć postać ciągu akceptowanych przez niego poleceń. Polecenia te są bardzo proste – można je podzielić na następujące grupy:

- polecenia przepisywania danych pomiędzy procesorem a pamięcią operacyjną,
- polecenia wykonania podstawowych operacji arytmetycznych (+, -, *, /),
- polecenia umożliwiające podejmowanie decyzji i przechodzenie do innych fragmentów programu,
- polecenia współpracy z urządzeniami zewnętrznymi.

O programie będącym ciągiem poleceń procesora mówi się, iż jest on zapisany w *języku wewnętrznym* lub w *języku asemblera*. Na szczęście programiści bardzo rzadko muszą zapisywać programy bezpośrednio w języku wewnętrznym – dla ich wygody wprowadzono znacznie dogodniejszy sposób zapisu programów, za pomocą tzw. *języków proceduralnych* (zwanych dawniej *językami programowania wysokiego poziomu*). Ponieważ procesor może wykonywać wyłącznie program zapisany w języku wewnętrznym konieczne jest przekształcenie na tę postać programów zapisanych w językach wyższego poziomu.



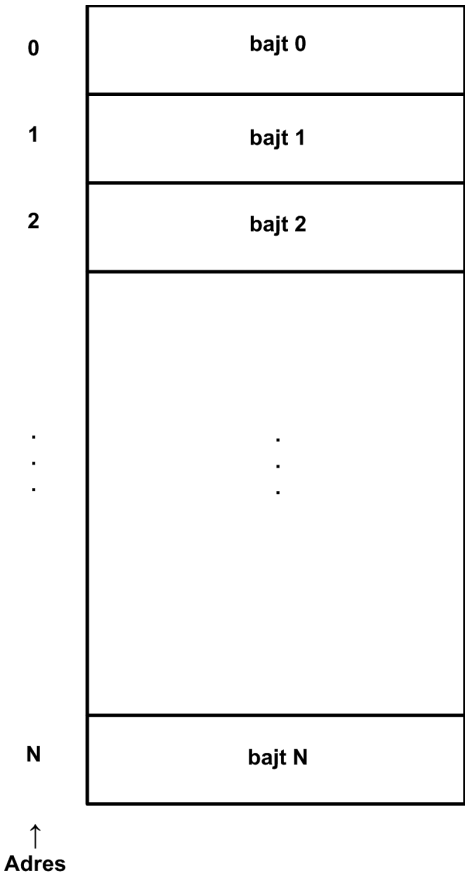

Rys. 1.5. Podstawowy cykl pracy komputera

Programy będące ciągami poleceń i dane przechowywane są w pamięci operacyjnej. Z tej pamięci procesor pobiera więc do wykonania kolejne polecenie i następnie je wykonuje. Podczas wykonywania polecenia z pamięci operacyjnej mogą być pobierane odpowiednie dane (rys. 1.5).

### 1.3.2. Pamięć operacyjna

*Pamięć operacyjna* komputera służy do przechowywania programów (zapisanych jako ciągi poleceń) oraz danych. Przymiotnik operacyjna oznacza, iż procesor posiada do niej bezpośredni dostęp. Przesyłanie danych pomiędzy procesorem i pamięcią operacyjną powinno odbywać się z maksymalną, możliwą do osiągnięcia prędkością, ponieważ prędkość ta ma decydujący wpływ na ostateczną prędkość pracy komputera.

Pamięć operacyjna składa się z ponumerowanych *komórek*, służących do przechowywania ciągów binarnych (rys. 1.6). Od wielu już lat obowiązuje niepisana umowa, iż w pojedynczej komórce pamięci przechowywany jest ciąg binarny o długości 8 bitów, zwany *bajtem*. Stąd popularnie mówi się o pamięci złożonej z bajtów. Numer komórki (bajtu) nazywany jest jej *adresem*, współczesne komputery wyposażane są w pamięci operacyjne o wielkości Gigabajtów (1 Gigabajt = $1024^3$ bajtów).




Rys. 1.6. Budowa pamięci operacyjnej

### 1.3.3. Urządzenia zewnętrzne

Liczne rodzaje urządzeń zewnętrznych dołączanych do komputera można podzielić na następujące grupy:

- urządzenia do ręcznego wprowadzania danych,
- urządzenia do wyprowadzania danych,
- pamięci zewnętrzne,
- urządzenia do przesyłania danych do/z sieci komputerowej oraz do/z innych urządzeń elektronicznych
- urządzenia specjalistyczne.

W ramach prezentowanego w tym podręczniku kursu programowania przedstawiony zostanie sposób korzystania z urządzeń należących do pierwszych trzech z wymienionych grup. Będzie to *klawiatura alfanumeryczna*, *monitor ekranowy* i *pamięć dyskowa* (rozdział 9.).

## 1.4. Reprezentacja danych w pamięci komputera

W pamięci komputera (operacyjnej lub zewnętrznej) mogą być przechowywane dane różnych rodzajów, np.:

- liczby,
- teksty,
- obrazy,
- dźwięki.

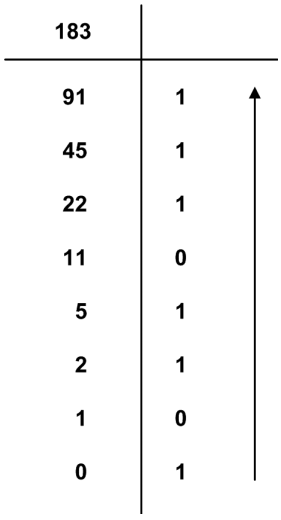
Ale, jak już powiedziano, pamięć komputera przechowuje wyłącznie ciągi zerojedynkowe – stąd wszystkie te rodzaje danych muszą zostać zapisane właśnie za pomocą takich zerojedynkowych ciągów. Przedstawiony teraz zostanie sposób reprezentowania w pamięci komputera liczb i tekstów (sposoby reprezentowania obrazów i dźwięków omawiane są w podręcznikach dotyczących technik multimedialnych).

### 1.4.1. Liczby

W przypadku liczb całkowitych należy podać sposób przekształcenia (konwersji) liczb zapisanych w powszechnie stosowanym systemie dziesiętnym na równe co do wartości liczby zapisane w systemie binarnym i odwrotnie.

#### Konwersja dziesiętno-binarna

Aby dokonać tej konwersji należy całkowitą, dodatnią liczbę dziesiętną dzielić przez 2 i zapisywać w następnej linii wartość całkowitą ilorazu oraz resztę. Proces ten kończy się po osiągnięciu ilorazu równego 0. Odczytując reszty w kolejności od ostatniej wyznaczonej do pierwszej otrzymuje się liczbę binarną o wartości równej wyjściowej liczbie dziesiętnej (rys. 1.7).




Rys. 1.7. Konwersja dziesiętno-binarna

### Konwersja binarno-dziesiętna

Dla dokonania zamiany całkowitej, dodatniej liczby binarnej na liczbę dziesiętną wykorzystuje się wzór na wartość liczby zapisanej w wagowym systemie pozycyjnym:

$$W = a_0 p^0 + a_1 p^1 + a_2 p^2 + ... \qquad [1]$$

gdzie:

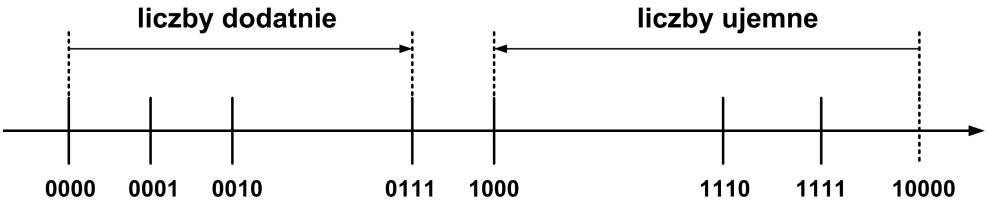
$W$ – wartość liczby,

$a_i$ - wartość i-tej cyfry,

$p$ – podstawa systemu zapisu.

Dla systemu dziesiętnego podstawa $p$ = 10, a kolejne potęgi $p$ to 1, 10, 100,... W przypadku systemu binarnego $p$ = 2, a kolejne potęgi $p$ to 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, ... Aby więc liczbę całkowitą binarną B = 10110111 zamienić na liczbę D o takiej samej wartości, zapisaną w systemie dziesiętnym, należy kolejne cyfry liczby binarnej i kolejne wartości potęg binarnych podstawić do wzoru [1] (pamiętając, że cyfra znajdująca się na prawym końcu liczby binarnej ma indeks 0):

D = 1*1 + 1*2 + 1*4 + 0*8 + 1*16 + 1*32 + 0*64 + 1*128
czyli D = 183.




Rys. 1.8. Uzupełnienie do 2

Liczby całkowite ze znakiem (ujemne albo dodatnie) przechowywane są w pamięci komputera w formacie zwanym *uzupełnieniem do 2* (*Uzp2*). Na rys. 1.8 przedstawiono ten format dla liczb 4-bitowych. Jak widać kolejne liczby dodatnie o wartościach dziesiętnych od 0 do 7 zapisywane są w sposób naturalny z wykorzystaniem 3 bitów (najstarszy bit jest równy 0). Liczby ujemne o wartościach dziesiętnych od –1 do –8 zapisywane są licząc od wartości binarnej 10000 (ta wartość reprezentuje w tym przypadku 2, występujące w nazwie formatu). Dla liczb ujemnych najstarszy bit ma zawsze wartość 1. W tabeli 1.1 przedstawiono ciągi binarne, odpowiadające liczbom o wartościach dziesiętnych od –8 do +7. Taki sposób reprezentowania liczb dodatnich i ujemnych wydaje się skomplikowany – jest on jednak bardzo wygodny przy realizacji dodawania i odejmowania tych liczb.

W programowaniu stosowane są również liczby o podstawie 8 (*liczby ósemkowe* lub *oktalne*) i 16 (*liczby szesnastkowe* lub *heksadecymalne*). Przyczyną wykorzystywania takich liczb jest łatwość dokonania ich konwersji na liczbę binarną i odwrotnie. Dla obliczenia binarnej reprezentacji dowolnej liczby całkowitej ósemkowej/szesnastkowej należy każdą jej cyfrę zamienić na 3/4 bitowy odpowiednik (tab. 1.2). Konwersja liczby binarnej na postać ósemkową/szesnastkową polega na podziale ciągu binarnego na segmenty 3/4 bitowe i zastąpienie każdego segmentu odpowiednią cyfrą ósemkową/szesnastkową. Cyfry ósemkowe to cyfry 0 ... 7. Dla zapisu cyfr szesnastkowych (których jest 16) wykorzystuje się cyfry 0 ... 9 oraz litery A ... F (stosowane są też litery a ... f).

Tabela 1.1. Liczby czterobitowe dodatnie i ujemne

| | | | |
|---|---|---|---|
| 0 | 0000 | –8 | 1000 |
| +1 | 0001 | –7 | 1001 |
| +2 | 0010 | –6 | 1010 |
| +3 | 0011 | –5 | 1011 |
| +4 | 0100 | –4 | 1100 |
| +5 | 0101 | –3 | 1101 |
| +6 | 0110 | –2 | 1110 |
| +7 | 0111 | –1 | 1111 |

Tabela 1.2. Cyfry szesnastkowe

ósemkowe

| | | | |
|---|---|---|---|
| 0 | 0000 | 8 | 1000 |
| 1 | 0001 | 9 | 1001 |
| 2 | 0010 | A | 1010 |
| 3 | 0011 | B | 1011 |
| 4 | 0100 | C | 1100 |
| 5 | 0101 | D | 1101 |
| 6 | 0110 | E | 1110 |
| 7 | 0111 | F | 1111 |

Liczby niecałkowite (rzeczywiste) przechowywane są w pamięci komputera jako tzw. *liczby zmiennopozycyjne.* Wartość liczby przechowywana jest w postaci wykładniczej jako

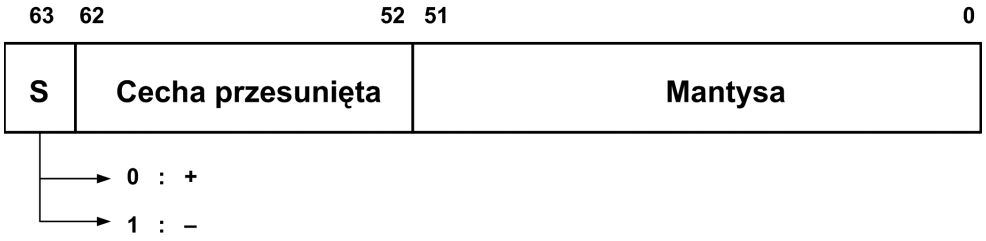
$$\pm m * 2^{\pm c}$$

gdzie:

m – *mantysa* liczby

c – *cecha* liczby

Mantysa zapisywana jest jako binarny ułamek właściwy, cecha jako liczba całkowita binarna (tzw. cecha przesunięta). Istnieje kilka rodzajów liczb zmiennopozycyjnych, różniących się zakresami przechowywanych wartości. Format najpopularniejszej liczby zmiennopozycyjnej (liczba podwójna – *double*) przedstawiono na rys. 1.9. Formaty te są zdefiniowane w zaleceniu IEEE 754.




Rys 1.9. Format zapisu liczby zmiennopozycyjnej typu *double*

## 1.4.2. Teksty

Aby utworzyć reprezentację tekstu w pamięci komputera należy zdefiniować sposób reprezentowania każdego znaku graficznego, występującego w alfabecie języka, w którym tekst ten został zapisany. Na przykład dla języka polskiego musi istnieć sposób jednoznacznej reprezentacji liter a ÷ ź, A ÷ Ź oraz cyfr 0 ÷ 9 (traktowanych jako znaki graficzne), znaków przestankowych i różnych znaków specjalnych (np. znaku @ ). Ponieważ w pamięci komputera przechowywane są jedynie ciągi binarne, których najprostszą interpretacją są liczby całkowite, należy każdemu znakowi graficznemu alfabetu przypisać unikatową liczbę całkowitą. Takie przypisanie nosi nazwę *kodowania*, liczba związana z danym znakiem graficznym zwana jest jego *kodem.* Pełen wykaz kodów dla znaków graficznych alfabetu to *tabela kodowania* lub po prostu *kod*.

Opracowano wiele tabel kodowania dla znaków graficznych (znaków alfanumerycznych) należących do alfabetów języków naturalnych. Do końca lat 80-tych XX wieku powszechnie stosowanym kodem był kod ASCII (Dodatek A). Kod ten przypisuje każdej literze, cyfrze i popularnym znakom specjalnym ciągi binarne

o długości 1 bajta. Wynika z tego, iż za pomocą tego kodu można reprezentować 256 różnych znaków graficznych. Nie jest to liczba wystarczająca, biorąc pod uwagę wielość znaków specjalnych i liter diakrytyzowanych (np. ą, ł, Ź, ä, õ, Ę ) występujących w różnych językach naturalnych. Stąd w kodzie ASCII wyróżniono 128 znaków podstawowych (litery łacińskie, cyfry, znaki sterujące i podstawowe znaki specjalne) – znakom tym przypisano kody od 0 do 127. Pozostałe znaki graficzne (w tym polskie znaki diakrytyzowane) podzielono na grupy po 128 znaków – do kodowania każdej takiej grupy użyto takiego samego zakresu kodów od 128 do 255. Rezultatem wprowadzenia takiego rozwiązania jest niejednoznaczność kodowania. Nie wiadomo np. jakiemu znakowi graficznemu odpowiada kod 205 – trzeba wiedzieć do jakiej grupy znaków go odnieść, czyli jaką *stronę kodową* zastosować. Problem ten istnieje do chwili obecnej – zdarza się niekiedy, że np. zamiast polskiej litery *ó* wyświetlany jest jakiś dziwny symbol. Większość programów edycyjnych umożliwia wybranie sposobu kodowania, czyli wskazanie strony kodowej, która ma być stosowana do przetwarzanego tekstu. Poprawne wyświetlanie i drukowanie polskich liter zapewnia np. kodowanie *Eastern Europe*.

Niedostatki kodu ASCII spowodowały opracowanie w latach 90-tych XX wieku nowego, jednoznacznego standardu kodowania, zwanego *Unicode*. Kodowanie to przewiduje, w maksymalnym wariancie, stosowanie do reprezentowania znaków graficznych ciągów 32-bitowych. Dzięki temu można zakodować jednoznacznie ponad 4 mld znaków graficznych. Jest to liczba wystarczająca dla objęcia jednoznacznym kodowaniem wszystkich znaków graficznych należących do wszystkich języków naturalnych jakie istnieją na naszej planecie (włączając w to znaki języka chińskiego i znaki języków martwych) oraz tysięcy znaków specjalnych.

Nie można jednak było z dnia na dzień zarzucić stosowania kodu ASCII – za pomocą tego kodowania do końca XX wieku zapisano olbrzymie ilości danych, zostało ono też zastosowane w milionach różnych programów. Opracowano więc różne warianty systemu Unicode. Najprostszy z nich i najczęściej stosowany to wariant UTF-8. Zakłada on, że 128 podstawowych znaków graficznych jest kodowanych tak, jak w kodzie ASCII, za pomocą kodu jednobajtowego. Pozostałe znaki wymagają kodów o długości 2, 3 lub 4 bajtów. Dzięki takiemu rozwiązaniu można nadal korzystać z danych zapisanych w kodzie ASCII (w zakresie znaków podstawowych) i równocześnie kod jest jednoznaczny, czyli na podstawie liczbowej wartości kodu można zawsze wskazać znak graficzny, który on reprezentuje.

# 2. Programowanie

## 2.1 Zmienne i struktury danych

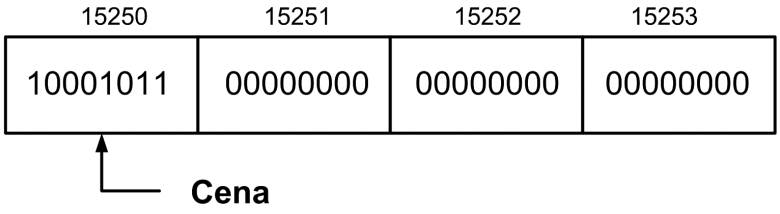
Program, który ma zostać wykonany przez komputer, jest zapisem algorytmu rozwiązującego pewien problem dotyczący przetwarzania danych. Dane w ramach programu są reprezentowane przez:

- zmienne,
- struktury danych.

*Zmienne* służą do reprezentowania danych pojedynczych, nie powiązanych ze sobą żadną relacją porządkującą. Każda zmienna może być uważana za parę:

{ *identyfikator* , *wartość* }

gdzie *identyfikator* jest unikatową nazwą zmiennej, czyli pojedynczym wyrazem danego języka. *Wartość zmiennej* zależy od jej rodzaju – większość zmiennych to zmienne liczbowe (numeryczne), ich wartościami są liczby całkowite lub zmiennopozycyjne. Identyfikatorem zmiennej programista posługuje się w programie, natomiast wartość zmiennej jest przechowywana w pamięci komputera za pomocą odpowiedniej liczby bajtów. I tak np. jeżeli identyfikatorowi `Cena` zostanie w programie nadana wartość 139, to w pamięci komputera zarezerwowane zostaną 4 bajty dla przechowywania binarnej postaci liczby 139 (rys. 2.1, dlaczego akurat 4 bajty zostanie wyjaśnione w rozdziale 3).




Rys. 2.1. Identyfikator i wartość zmiennej

Adres pierwszego z tej czwórki bajtów zostaje powiązany z identyfikatorem `Cena` i kolejne zmiany wartości tej zmiennej będą powodowały zmianę ciągu binarnego, w tych bajtach przechowywanego. Jak widać na rysunku 2.1 najmłodsza część ciągu binarnego reprezentującego liczbę dziesiętną 139 została zapisana w bajcie

o najmniejszym adresie (w bajcie najbliższym początku pamięci). Taki sposób zapisu danych liczbowych nosi nazwę mało-końcowego (ang. *little endian*) i jest powszechnie stosowany przy zapisie liczb w pamięci komputera. Operacja nadania nowej wartości zmiennej jest zapisywana na różne sposoby, na razie będzie w tym celu używany znak strzałki w lewo.

```
Cena ← 139
. . . . . .
Cena ← 99
```

Nadanie nowej wartości zmiennej powoduje usunięcie jej poprzedniej wartości. W programach często występują zapisy o postaci:

```
Cena ← Cena + 14
```

Zapis ten nie zawiera sprzeczności, jak to by się mogło wydawać na pierwszy rzut oka. Wskazuje on, że należy odczytać dotychczasową wartość zmiennej Cena (czyli 99), powiększyć ją o 14 i wynik zapisać jako nową wartość zmiennej Cena (113).

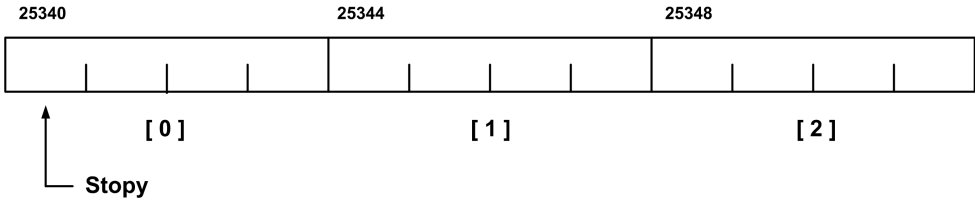
Jeżeli pomiędzy pewną liczbą danych występuje powiązanie kolejnościowe lub powiązanie przynależnościowe, to taki zestaw danych zapisuje się za pomocą *struktury danych.*

Najprostszą strukturą danych jest *tablica*, zawierająca ciąg uporządkowanych kolejno wartości takiego samego typu. Wartości te to *elementy tablicy.* W najprostszym przypadku (*tablicy jednowymiarowej*) tworzą one ciąg ponumerowany za pomocą kolejnych liczb całkowitych dodatnich. Numeracja ta rozpoczyna się od wartości 0. Każda tablica posiada unikatowy identyfikator. Numer elementu tablicy (czyli jego *indeks*) zapisywany jest w nawiasach prostokątnych.

```
Wymiary [ 0 ]          // początkowy element tablicy Wymiary
Wymiary [ 15 ]         // element tablicy Wymiary o indeksie 15
Wymiary [ ind_1 ]      // element tablicy Wymiary o indeksie
                       // równym wartości zmiennej ind_1
```

Jeżeli w programie użyta zostanie np. tablica Stopy zawierająca 3 elementy, będące liczbami całkowitymi, to w pamięci komputera zarezerwowany zostanie spójny obszar o długości 3 * 4 = 12 bajtów dla przechowywania wartości elementów tej tablicy (rys. 2.2). Z identyfikatorem Stopy powiązany zostanie adres początkowego bajtu, początkowego elementu tej tablicy (czyli elementu Stopy[ 0 ] ).

Tablice jednowymiarowe są też wykorzystywane do przechowywania danych tekstowych – elementy takiej tablicy to kody znaków graficznych pewnego alfabetu. Niekiedy, dla zaznaczenia końca tekstu, jako ostatni dodaje się element o wyróżnionej wartości (punkt 3.7).

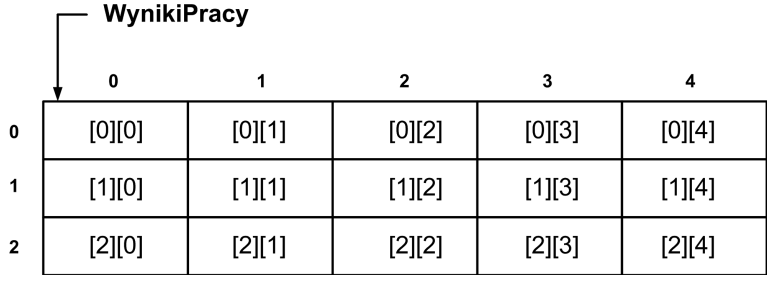



Rys. 2.2. Tablica jednowymiarowa

Tablice dwuwymiarowe są reprezentacją powszechnie stosowanych tabelek złożonych z wierszy i kolumn (i również znanych z algebry *macierzy*). Aby wskazać element takiej tablicy należy więc podać numer wiersza (liczony od 0) i numer kolumny (również liczony od 0) w kolejności: numer_wiersza, numer_kolumny.

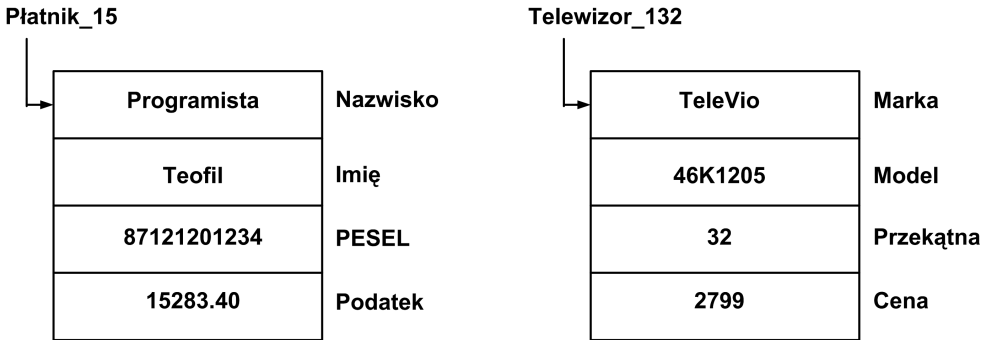
```
Wynagrodzenia [12] [15]      // wiersz numer 12, kolumna numer 15
Wynagrodzenia [x] [y]        // wiersz o numerze będącym wartością x
                             // kolumna o numerze będącym wartością y
```

Gdy w programie wystąpi np. tablica Wyniki_Pracy o 3 wierszach i 5 kolumnach, której elementami są liczby całkowite, to w pamięci komputera rezerwowany jest spójny obszar o długości 3 * 5 * 4 = 60 bajtów (rys. 2.3). Z identyfikatorem WynikiPracy powiązany zostanie adres początkowego bajtu elementu z początkowego wiersza i początkowej kolumny (czyli elementu WynikiPracy [ 0 ] [ 0 ] ).




Rys. 2.3. Tablica dwuwymiarowa

Tablice stosowane są w przypadkach, gdy dane powiązane są kolejnościowo (w jednym lub dwu wymiarach). Wszystkie elementy tablicy muszą być takiego samego typu. W procesie przetwarzania danych często występują również dane powiązane przynależnościowo, to jest dane różnych typów opisujące pewien przedmiot czy pewną osobę. Struktura danych, grupująca dane będące opisem przedmiotu czy osoby, nosi nazwę *rekordu* lub po prostu *struktury*. Elementy struktury to pola, każde pole może być innego typu (rys. 2.4).




Rys. 2.4. Struktury

Struktura Płatnik_15 składa się z 4 pól. Pierwsze dwa z nich (Nazwisko i Imię) to pola tekstowe (czyli jednowymiarowe tablice kodów znaków graficznych), pole PESEL zawiera liczbę całkowitą, a pole Podatek liczbę zmiennopozycyjną. Podobnie pola struktury Telewizor_132 zawierają dane różnych typów.

Istnieje wiele innych rodzajów struktur danych, niektóre z nich zostaną przedstawione w punktach 6.3 i 6.4 .

## 2.2. Zapisywanie algorytmów

### 2.2.1. Schematy blokowe

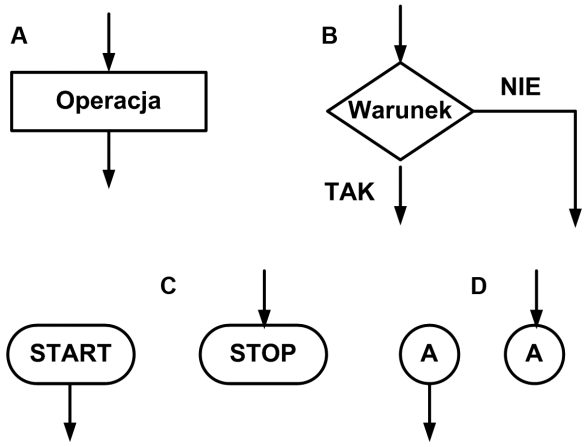
Jednym z prostszych sposobów zapisywania algorytmów (czyli pisania programów) jest użycie do tego celu notacji graficznej, zwanej językiem *schematów blokowych* (lub *siecią działań*). Taki zapis umożliwia nadanie algorytmowi trwałej formy (czyli formy programu), nie umożliwia on jednak przekazania tak przygotowanego programu do wykonania przez komputer (pomijając niektóre systemy edukacyjne). Niemniej rysowanie schematów blokowych pozwala na zapoznanie się z podstawowymi składnikami, z których budowane są programy.

Dla potrzeb tego podręcznika wystarczające są 4 rodzaje bloków (rys. 2.5). Pełen zestaw bloków dla języka schematów blokowych definiuje norma PN-75/E-01226.

Blok A zawiera opis dowolnej operacji przetwarzania danych. Może to być operacja bardzo prosta (np. Numer ← Numer + 1) lub bardzo złożona (np. Wyznacz rozwiązania równania różniczkowego R_1). Ważne jest, że blok operacji zawsze posiada jedno wejście i jedno wyjście.

Blok B to blok warunkowy, czyli tzw *predykat,* umożliwiający zapisanie w programie operacji podejmowania decyzji. Posiada on jedno wejście i dwa wyjścia. Jeżeli warunek wpisany w bloku warunkowym jest prawdziwy, to nastąpi przejście wyjściem TAK, przeciwnie, gdy warunek nie jest prawdziwy, wyjściem NIE. Położenie graficzne wyjść TAK/NIE można zmieniać, ale zawsze musi istnieć jedno wejście do bloku warunkowego i dwa z niego wyjścia. Przykładowy warunek może mieć postać

Alfa > 0. Jeżeli po wejściu do takiego bloku zmienna Alfa ma wartość większą od 0, to nastąpi wyjście drogą TAK, przeciwnie, gdy Alfa jest mniejsza lub równa 0, wybrana zostanie droga NIE.




Rys. 2.5. Bloki podstawowe

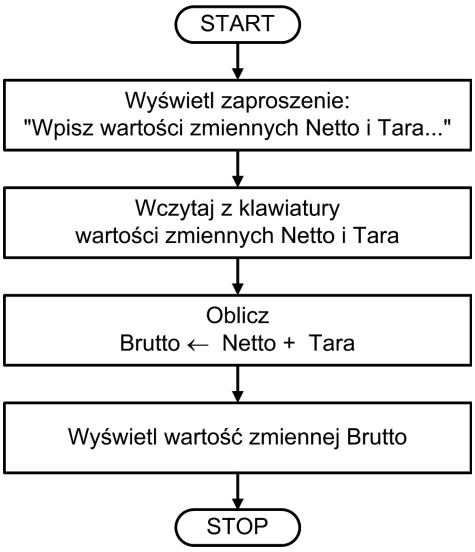
Bloki z grupy C służą do rozpoczynania i kończenia schematu blokowego. W każdym schemacie powinien wystąpić dokładnie jeden blok START i przynajmniej jeden blok STOP.

Ostatnia grupa bloków (D) zawiera proste bloki służące do łączenia części schematu blokowego, które znajdują się na różnych stronach.

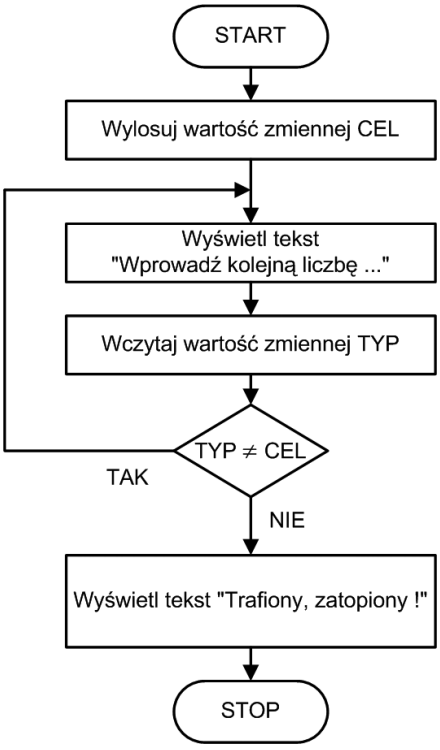
Budowanie schematu blokowego będącego zapisem pewnego algorytmu przetwarzania danych rozpoczyna się od bloku START. Następnie można wprowadzać i łączyć ze sobą dowolną liczbę bloków operacji i bloków warunkowych. Koniec programu zapisanego jako schemat blokowy następuje po osiągnięciu bloku STOP.

Rysunek 2.6 zawiera przykład konkretnego programu obliczającego wartość Brutto ← Netto + Tara. Przygotowując taki program należy uczynić go uniwersalnym, czyli takim, który wyliczy wartość Brutto do dowolnych, aktualnych wartości Netto i Tara. Konkretne liczbowe wartości aktualne, które mają zostać wykorzystane w danym wykonaniu programu, powinny zostać podane przez użytkownika podczas wykonywania programu. Program ten ma charakter liniowy – nie występują w nim żadne bloki warunkowe.

Jak już powiedziano programy zapisane za pomocą schematów blokowych nie bardzo nadają się do wykonania przez komputer. Program tak zapisany można jedynie wykonać ręcznie, przechodząc od startu do stopu, wykonując kolejne operacje (obliczenia) i notując na kartce kolejne wartości zmiennych. Takie wykonanie programu jest działaniem wysoce kształcącym, pozwala na dokładne zrozumienie procesu przetwarzania danych.




Rys. 2.6. Program liniowy




Rys. 2.7. Program z rozgałęzieniem

Na rysunku 2.7 pokazano prosty program realizujący grę Zgadnij_Liczbę. Po wylosowaniu wartości, którą gracz powinien zgadnąć, jest on proszony o podanie swojej kandydatki. W bloku warunkowym następuje sprawdzenie, czy gracz wprowadził poprawną wartość, jeżeli nie, to następuje przejście z powrotem do wyświetlania zaproszenia do dalszej gry. Gdy TYP = CEL następuje wyświetlenie informacji o trafieniu i koniec programu.

### 2.2.2. Języki programowania

Aby program mógł zostać wykonany przez komputer musi on zostać zapisany za pomocą odpowiedniego języka programowania. W rozdziale 1. wspomniano o języku wewnętrznym, w którym program ma postać ciągu poleceń realizowanych bezpośrednio przez procesor. Znacznie wygodniejsze jest użycie do zapisu programu jednego z licznych *języków proceduralnych* (zwanych dawniej *językami programowania wysokiego poziomu* – język wewnętrzny jest językiem niskiego poziomu). Języki takie należą do języków sztucznych o dokładnie zdefiniowanej składni. Pozwalają one między innymi na:

- deklarowanie zmiennych i struktur danych, które mają być użyte w programie,
- proste zapisywanie wyrażeń arytmetycznych i logicznych,
- stosowanie instrukcji sterujących, zawierających proste słowa języka angielskiego, do określania kolejności wykonania operacji (np. while, if ),
- podział programu na podprogramy i ich wywoływanie,
- korzystanie z obszernych bibliotek zawierających między innymi podprogramy obliczania wartości funkcji arytmetycznych czy trygonometrycznych, podprogramy umożliwiające korzystanie z urządzeń zewnętrznych itp.

Języki proceduralne zaczęto definiować w latach 50-tych XX wieku (pierwszy masowo stosowany język programowania nosi nazwę FORTRAN). Spośród setek opracowanych do dnia dzisiejszego języków programowania tylko kilka jest na większą skalę stosowanych w praktyce. Wymienić tu można:

- języki programowania strukturalnego: C, Pascal, Ada, PL1,
- języki programowania obiektowego: C++, C#, Object Pascal, Java
- języki opisu dokumentów i stron WWW: HTML, XML, PHP, JAVA SCRIPT.

Ponadto zdefiniowano wiele języków specjalistycznych, np. LISP – do przetwarzania struktur listowych, SIMULA – do zapisu programów symulacyjnych itp.

Wyjaśnienia wymagają jeszcze terminy *programowanie strukturalne* i *programowanie obiektowe*. Zasadę programowania strukturalnego wprowadzono dla ograniczenia chaosu, jaki łatwo wprowadzić do programu, stosując przejścia pomiędzy jego dowolnymi miejscami. Beztroscy programiści tworzyli programy bardzo zagmatwane, sprawdzenie ich poprawności było często prawie niemożliwe. Zasada programowania strukturalnego ogranicza tę samowolę, przejście pomiędzy różnymi

miejscami programu może odbywać się wyłącznie w sposób ustandaryzowany, za pomocą struktur sterowania (punkt 2.3.3). Wykazano, iż dowolny algorytm można zapisać stosując zasady programowania strukturalnego.

Koncepcja programowania obiektowego pojawiła się w latach 60-tych, a na szeroką skalę została wprowadzona w latach 90-tych XX wieku. Metoda ta zakłada, że w programie tworzone będą *obiekty* będące odwzorowaniem odpowiednich wycinków rzeczywistości. Obiekt grupuje pewną liczbę danych i struktur danych oraz zestaw operacji dane te przetwarzających. Ograniczenia, które można w programowaniu obiektowym narzucić na możliwości dostępu do poszczególnych grup danych, pozwalają na tworzenie programów o znacznie wyższej odporności na błędy powstające podczas ich wykonywania. Języki programowania obiektowego są obecnie najczęściej stosowanymi językami programowania.

### 2.2.3. Poprawność programów

Przyjmując, że algorytm jest poprawny, czyli rzeczywiście umożliwia uzyskanie poprawnych rozwiązań danego problemu przetwarzania danych, należy zastanowić się, czy poprawny jest też program będący tego algorytmu zapisem. Programowanie jest procesem, w którym łatwo popełnić błędy, a z drugiej strony wykazanie poprawności programu nie jest zadaniem łatwym. Dla prostych programów można stosować formalne metody dowodzenia poprawności, są one jednak całkowicie niepraktyczne w zastosowaniu do wielkich programów profesjonalnych. Jedyne praktyczne możliwości oceny poprawności programu to przeglądanie tekstu programu (najlepiej dokonywane przez osobę nie będącą tego programu autorem) i staranne testowanie programu. Obydwie te metody są niestety zawodne. Osoba przeglądająca cudzy program dość szybko ulega zmęczeniu i nie dostrzega prostych błędów – sposób ten ma więc ograniczone zastosowanie. Podstawową metodą pozostaje więc testowanie programu, czyli wielokrotne jego wykonywanie dla najróżniejszych zestawów danych wejściowych. Proces testowania dużych programów jest realizowany zazwyczaj przez specjalną grupę pracowników (testerów). Na początku procesu testowania błędy wykrywane są (i usuwane) dość często, później czas potrzebny na wykrycie kolejnego błędu rośnie. Jeżeli przez pewien okres czasu (np. dwa tygodnie) testowanie nie doprowadziło do wykrycia żadnego błędu, to przyjmuje się (być może optymistycznie), że program jest poprawny. Do procesu testowania włączani są często przyszli użytkownicy tworzonego programu – producent udostępnia im tzw. wersję beta, licząc na współpracę w odnajdywaniu dalszych błędów. Bardzo nawet staranne testowanie nie prowadzi niestety do uzyskania pewności co do poprawności programu (nie jest dowodem poprawności). Przekonujemy się o tym na co dzień – sztandarowe produkty wybitnych firm informatycznych często wymagają poprawek już po krótkim czasie bardziej powszechnego użytkowania. Podobnie, w pozornie poprawnych programach, używanych od wielu lat, niespodziewanie pojawiają się błędy.

Uwagi te prowadzą do następujących wniosków:

- nigdy nie można mieć pewności, że dany program będzie działał poprawnie,
- programowanie jest działalnością bardzo odpowiedzialną, szczególnie gdy w grę wchodzi życie, zdrowie i pieniądze ludzi,
- programować należy z najwyższą starannością, przestrzegając zaleceń i dobrych praktyk wypracowanych przez *inżynierię oprogramowania*.

## 2.3. Podstawowe składniki programu

### 2.3.1. Deklarowanie danych i struktur danych

Dane występujące w rzeczywistości są w programie reprezentowane za pomocą zmiennych i struktur danych (punkt 2.1). Jeżeli program zapisywany jest jako schemat blokowy, to nie potrzeba uprzedzająco zgłaszać potrzeby użycia jakiejś zmiennej czy struktury danych. Pierwsze wystąpienie identyfikatora zmiennej czy struktury danych równoważne jest z wprowadzeniem jej do programu. Zasada ta stosuje się również do niektórych innych, beztrosko zdefiniowanych języków programowania.

W profesjonalnych językach programowania potrzebę użycia zmiennej czy też struktury danych należy zgłosić za pomocą odpowiedniej deklaracji. Korzystanie z niezadeklarowanych zmiennych czy struktur danych traktowane jest jako błąd.

Dla zmiennych deklaracja taka określa identyfikator i typ wartości zmiennej, co pozwala na zarezerwowanie odpowiedniej liczby bajtów pamięci komputera dla przechowywania wartości zmiennej.

```
int LiczbaKrzeseł = 15 ;
```

Deklaracja ta, zapisana w języku C, wskazuje, iż zmienna LiczbaKrzeseł będzie miała wartości całkowitoliczbowe, dodatnie lub ujemne, zapisane w pamięci komputera za pomocą 4 bajtów (parametry te wynikają z definicji użytego typu danych int). Początkowa wartość tej zmiennej będzie wynosiła 15.

W przypadku struktur danych deklaracja określa: identyfikator struktury danych, typ danych w niej występujących oraz liczbę tych danych.

```
double Przychody [ 12 ] = { 0.0 } ;
```

Tablica jednowymiarowa Przychody, której deklaracja również jest zapisana w języku C, będzie miała 12 elementów, każdy z nich to liczba zmiennopozycyjna, ośmiobajtowa. Wartości początkowe wszystkich tych elementów będą równe 0.0 .

### 2.3.2. Instrukcje

Instrukcje określają jakie operacje (obliczenia) i w jakiej kolejności zostaną zrealizowane podczas wykonania programu. Do ustalania kolejności wykonywania

operacji służą *instrukcje sterowania*, omówione szczegółowo w punkcie następnym. Pozostałe rodzaje instrukcji to najczęściej:

- instrukcja nadania wartości zmiennej lub elementowi struktury danych,
- instrukcja wywołania podprogramu,
- instrukcje wprowadzania i wyprowadzania danych.

Podstawową rolę pełni instrukcja nadawania nowej wartości zmiennej (*instrukcja przypisania*). Zwyczajowo identyfikator zmiennej jest zapisywany na początku (po lewej stronie) takiej instrukcji. Potem występuje operator przypisania wartości (jego postać zależy od konkretnego języka programowania), a po prawej stronie tego operatora występuje wyrażenie arytmetyczne lub logiczne, którego wartość w takcie wykonywania instrukcji zostaje obliczona i stanie się nową wartością zmiennej.

```
alfa = beta + 3 * delta – 1 ;                    // język C

delta := delta – beta + 5 ;                      // język Pascal
```

Jak widać w języku programowania C rolę operatora przypisania wartości pełni pojedynczy znak równości ( = ). W języku Pascal operator ten zbudowany jest z dwu znaków: dwukropka i znaku równości ( := ). Wartości zmiennych znajdujących się w wyrażeniu po prawej stronie operatora przypisania nie ulegają zmianie, o ile są to inne zmienne, niż zmienna, której nadawana jest nowa wartość.

```
int licznik = 5 ;                                // język C

int składnik = 7 ;

licznik = 10 * licznik + składnik ;              // linia 3
```

W podanym przykładzie zadeklarowane zostały dwie zmienne całkowitoliczbowe: licznik o początkowej wartości 5 i składnik o wartości 7. W trzeciej linii znajduje się instrukcja przypisania, która określa nową wartość zmiennej licznik. Wartość ta jest obliczana w sposób następujący: dotychczasowa wartość zmiennej licznik (czyli 5) zostaje pomnożona przez 10 i do wyniku zostanie dodana wartość zmiennej składnik (czyli 7). Ostatecznie wartość zmiennej licznik wyniesie 57, a wartość zmiennej składnik nie ulegnie zmianie.

Instrukcje wywołania podprogramu omówione zostaną w punkcie 2.3.4, natomiast instrukcje wprowadzania i wyprowadzania danych nie występują w języku C, który jest przedmiotem wykładu w tym podręczniku. W języku C operacje wprowadzania i wyprowadzania danych realizowane są za pomocą odpowiednich funkcji (podprogramów) bibliotecznych.

## 2.3.3. Struktury sterowania

Wykonywanie programu rozpoczyna się od wykonania jego początkowej instrukcji. Po wykonaniu n-tej instrukcji wykonywana jest instrukcja o numerze n+1. Od tej zasady kolejnego wykonywania instrukcji istnieje tylko jedno odstępstwo, dotyczące

*rozgałęzień*, dzięki którym można w programie podejmować decyzje co do dalszego jego przebiegu. Wykazano, że takie ograniczenie sposobu określania kolejności wykonywania instrukcji nie ogranicza uniwersalności programów – każdy algorytm można zapisać za pomocą programu, w którym reguły te są przestrzegane.

### 2.3.3.1. Rozgałęzienia

W punkcie dotyczącym schematów blokowych (2.2.1) omówiono blok warunkowy, który posiada jedno wejście i dwa wyjścia. Wybór wyjścia, którym dalej postępować będzie proces wykonywania programu zależny jest od spełnienia (lub nie spełnienia) warunku zapisanego w bloku warunkowym. Warunek jest spełniony, gdy reprezentujące go wyrażenie logiczne jest prawdziwe. W językach programowania wprowadzono prawie identyczne rozwiązanie zwane *instrukcją warunkową*. Definiuje się warunek – gdy jest spełniony, to wykonywany jest określony ciąg instrukcji. Gdy natomiast warunek nie jest spełniony, to albo wykonuje się inny, przewidziany na tę okoliczność ciąg instrukcji, albo nie wykonuje się żadnej dodatkowej instrukcji i następuje przejście do instrukcji następnej.

```
int baza, kwota = 0 ;              // język C
// .....                           // fragment A
// .....
if  ( baza > 127 )                 // linia N
      kwota = 745 ;
// .....                           // fragment B
// .....
if ( baza < 23 )                   // linia M
      kwota = 35 ;
else
      kwota = 89 ;
```

W przykładzie tym występuje zmienna baza, której wartość determinuje spełnienie czy też niespełnienie kolejnych warunków. Po wykonaniu instrukcji z fragmentu A, w linii N występuje instrukcja warunkowa, rozpoczynająca się od słowa if (jeżeli). Gdy w trakcie wykonania fragmentu A zmiennej baza zostanie nadana wartość większa od 127, to warunek z linii N będzie spełniony i wykonana zostanie instrukcja przypisania, nadająca zmiennej kwota wartość 745. Jeżeli natomiast po osiągnięciu linii N zmienna baza będzie miała wartość mniejszą lub równą 127, to przypisanie nowej wartości zmiennej kwota nie zostanie wykonane i nastąpi bezpośrednie przejście do fragmentu B. Tego, czy warunek z linii N będzie spełniony nie można przewidzieć z góry, ponieważ we fragmencie A program mógł pobrać dane zewnętrzne, od których konkretnych wartości mogła być uzależniona wartość nadana w tym fragmencie zmiennej baza.

Instrukcja warunkowa z linii M zawiera poza warunkiem i słowem if dodatkowe słowo else (przeciwnie), które rozgranicza instrukcje wykonywane, gdy warunek jest spełniony od instrukcji przewidzianych na wypadek, gdyby warunek spełniony nie był. Tak więc, zależnie od aktualnej wartości zmiennej baza zmienna kwota otrzyma wartość 35 albo 89.

### 2.3.3.2. Pętle

Bardzo często w algorytmie rozwiązującym jakiś problem występuje konieczność wielokrotnego wykonania takiej samej operacji dla kolejnych wartości pewnych danych. Przykładem może być problem obliczenia sumy liczb 1 ... N, czy też problem obliczenia podatku od wynagrodzeń pracowników, które to wynagrodzenia są zebrane w tablicy jednowymiarowej. Pierwszy z tych przykładów należy do grupy obliczeń krokowych, w których wykonanie kolejnego obliczenia wymaga znajomości wyniku obliczenia poprzedniego. Drugi przykład to przetwarzanie kolekcji danych (zawartych w pewnej strukturze danych) – każdy element kolekcji jest przetwarzany w taki sam sposób.

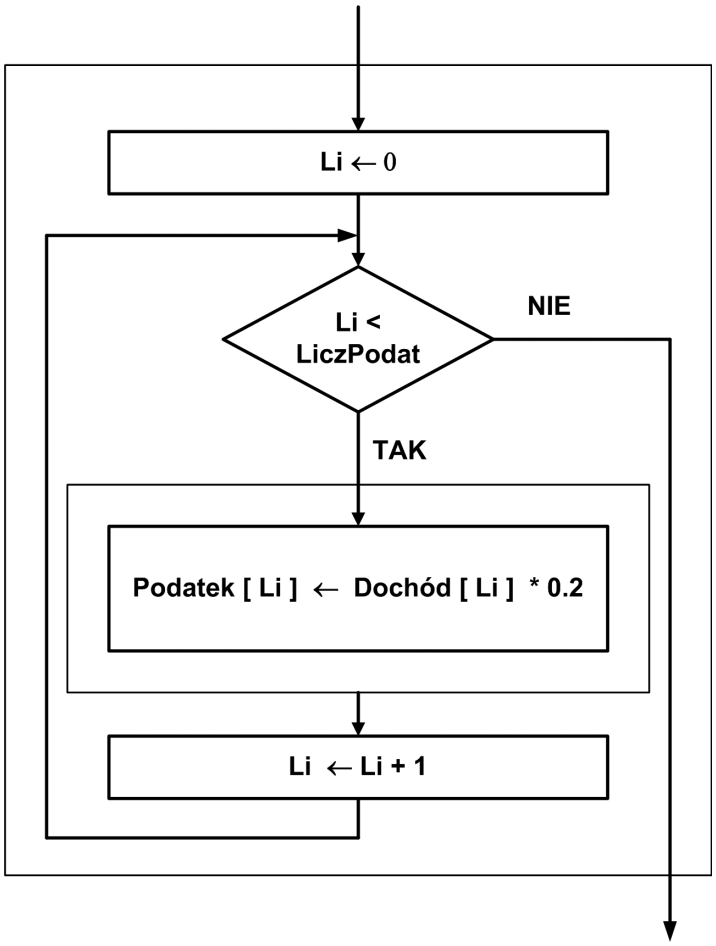
W programach, będących zapisami algorytmów, wielokrotne wykonanie takiej samej operacji jest realizowane za pomocą konstrukcji zwanej *pętlą*. W zapisie pętli istotne są:

- ustalenie warunków początkowych,
- definicja operacji powtarzanej w pętli (*treści pętli*),
- sformułowanie warunku zakończenia wykonywania pętli (*warunek wyjścia*).

Istnieją dwa podstawowe typy pętli:

- pętle wykorzystujące licznik – po każdym wykonaniu treści pętli licznik jest zmieniany, po czym sprawdza się, czy nie osiągnął wartości granicznej,
- pętle, których wykonanie kończy się po osiągnięciu przez przetwarzane dane zadanych wartości.

Na rysunku 2.8 przedstawiono schemat blokowy pętli obliczającej podatek od wynagrodzeń. Przed rozpoczęciem wykonywania tej pętli wartości wynagrodzeń wszystkich pracowników zapisane być muszą w tablicy Dochód. Ustalona też być musi wartość zmiennej LiczPodat, która określa liczbę elementów tej tablicy oraz liczbę elementów wynikowej tablicy Podatek. Dla każdego i-tego elementu tablicy Dochód obliczany jest stosowny podatek i jego wartość wpisywana jest jako i-ty element tablicy Podatek. Konstrukcja tej pętli rozpoczyna się od bloku ustalającego początkową wartość licznika Li (wynoszącą 0, ponieważ elementy tablic numerowane są od 0). Następnie, w bloku warunkowym sprawdza się, czy licznik Li ma jeszcze wartość mniejszą niż wartość zmiennej LiczPodat. Gdy ten warunek jest spełniony wykonuje się treść pętli (wyróżnioną odrębną ramką), składającą się w tym przypadku z jednego bloku zawierającego operację obliczenia podatku i wpisania jego wartości jako i-tego elementu tablicy Podatek. Kolejny element pętli to blok zwierający operację zmiany wartości licznika – w tym przypadku wartość licznika Li jest powiększana o 1. Treść pętli wykonywana jest wielokrotnie, ale cała konstrukcja pętli posiada tylko jedno wejście i jedno wyjście – pętla może więc być zawarta w pojedynczym bloku operacyjnym (co zostało na rys. 2.8 zaznaczone zewnętrznym obramowaniem z jednym wejściem i jednym wyjściem).




Rys. 2.8. Pętla wykorzystująca licznik

Ta sama pętla obliczająca podatki od wynagrodzeń może zostać zapisana w języku C w sposób następujący.

```
double Dochód [ 100 ] ;
double Podatek  [ 100 ] ;
int LiczPodat ;
int Li ;
// .....                                    // fragment A
for ( Li = 0 ; Li < LiczPodat ; Li = Li + 1)
      Podatek [ Li ] = Dochód [ Li ] * 0.2;  // 20%
// .....                                    //  fragment B
```

Po zadeklarowaniu tablic i zmiennych wykonywany jest fragment A, w którym ustalana jest wartość zmiennej LiczPodat i odpowiednie wartości wpisywane są do

tablicy Dochód. Pętla rozpoczyna się od słowa for (dla), następnie w nawiasach okrągłych zawarte są trzy składniki konstrukcji pętli (oddzielone średnikami). Pierwszy składnik to ustalenie wartości początkowej licznika. Składnik drugi to wyrażenie logiczne zwane warunkiem wyjścia – zakończenie wykonywania pętli następuje, gdy wyrażenie to będzie miało wartość *fałsz* (nieprawda, że). Ostatni składnik to instrukcja określająca sposób zmiany wartości licznika. W następnej linii znajduje się treść pętli. Wykonanie takiej pętli przebiega następująco:

a) ustalenie wartości początkowej licznika,
b) sprawdzenie warunku wyjścia, gdy wyrażenie ma wartość *fałsz*, następuje opuszczenie pętli i przejście do fragmentu B,
c) wykonanie treści pętli,
d) zmiana wartości licznika,
e) przejście do punktu b.

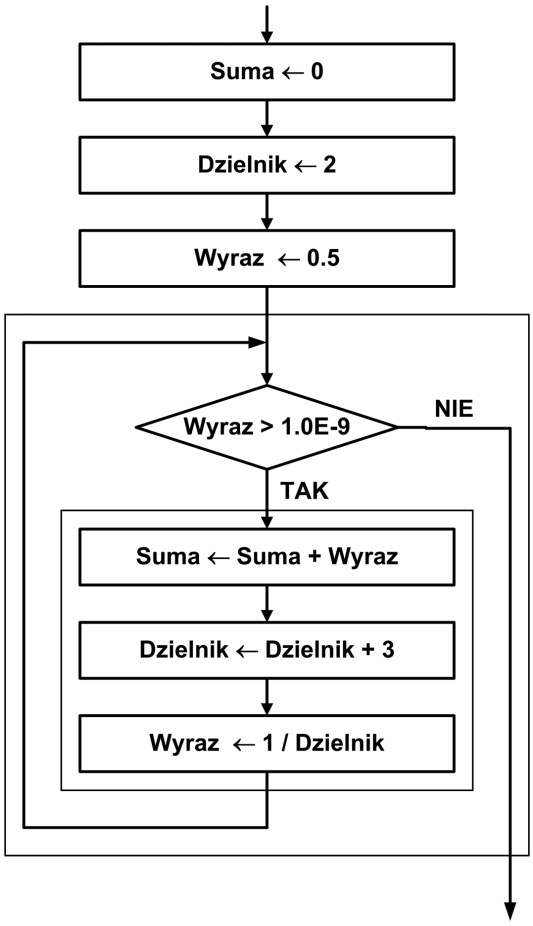
W prezentowanym przykładzie treść pętli była prosta – sprowadzała się do pojedynczej operacji. W bardziej złożonych programach treść pętli może być bardzo rozbudowana, w szczególności w treści pętli może być zawarta (*zagnieżdżona*) następna pętla. Zawsze jednak treść pętli musi dać się sprowadzić do pojedynczego bloku operacyjnego, posiadającego jedno wejście i jedno wyjście. Przestrzeganie tej reguły jest podstawowym wymogiem programowania strukturalnego.

Drugi rodzaj pętli to *pętla dopóki*, w której nie przewiduje się bezpośredniego stosowania licznika wykonań. W warunku wyjścia takiej pętli sprawdza się natomiast, czy przetwarzane zmienne (jedna lub wiele) osiągnęły odpowiednie wartości (*wartości graniczne*). Jako przykład może posłużyć program obliczający sumę wyrazów szeregu:

$$S = \frac{1}{2} + \frac{1}{5} + \frac{1}{8} + \frac{1}{11} + \cdots$$

Sumowanie ma być prowadzone dopóty, dopóki kolejny wyraz będzie większy od $10^{-9}$.

Pierwsze trzy bloki przedstawione na rys. 2.9 zawierają operacje ustalające warunki początkowe wykonania pętli. Sama pętla (zawarta w dodatkowym obramowaniu) rozpoczyna się od sprawdzenia warunku wyjścia – gdy wyrażenie logiczne reprezentujące ten warunek ma wartość *fałsz*, to następuje zakończenie wykonywania pętli. W treści pętli (również ograniczonej dodatkowym obramowaniem) następuje dodanie aktualnej wartości wyrazu do sumy, zwiększenie dzielnika o 3 i obliczenie wartości kolejnego wyrazu ciągu.




Rys. 2.9. Pętla dopóki

W języku C pętla obliczająca sumę wyrazów rozważanego szeregu może zostać zapisana w sposób następujący.

```
double Suma, Dzielnik, Wyraz ;
Suma = 0 ;
Dzielnik = 2 ;
Wyraz = 0.5 ;
while ( Wyraz > 1.0E-9 )
{
      Suma = Suma + Wyraz ;
      Dzielnik = Dzielnik + 3 ;
      Wyraz = 1 / Dzielnik ;
}
// .....                          // fragment A
```

Pierwsze cztery linie programu zawierają deklarację zmiennych i przypisanie im wartości początkowych. Pętla rozpoczyna się od słowa while (dopóki), za którym w nawiasach okrągłych wpisane zostało wyrażenie logiczne reprezentujące warunek wyjścia. Zakończenie wykonywania pętli i przejście do fragmentu A programu nastąpi, gdy wartość zmiennej Wyraz stanie się mniejsza lub równa od $10^{-9}$ . Treść pętli składa się w tym przykładzie z trzech instrukcji, zostały one połączone w instrukcję złożoną, za pomocą nawiasów klamrowych. Kolejne instrukcje wykonują dodanie aktualnej wartości wyrazu do sumy, zwiększenie dzielnika o 3 i obliczenie następnej wartości wyrazu sumowanego ciągu.

### 2.3.4. Podprogramy

Modularyzacja programu, czyli jego podział na części, zwane podprogramami lub modułami, jest nieodzowna w przypadku nieco już większych programów. Istnieją dwa powody dokonywania takiego podziału. Po pierwsze, w ramach wyróżnionego podprogramu można zawrzeć operacje przetwarzające, które będą wielokrotnie stosowane w rożnych miejscach programu. Przykładem może być podprogram wyświetlający na ekranie tabelę. W każdym miejscu programu, w którym dane maja zostać zaprezentowane w formie tabeli, można będzie z takiego podprogramu skorzystać. Drugim, daleko ważniejszym powodem, jest potrzeba ograniczenia liczby instrukcji, które programista ma napisać w ramach pojedynczego modułu i których poprawność należy potem sprawdzić. Zweryfikowanie poprawności programu złożonego z tysięcy linii (a taką liczbę linii posiada wiele profesjonalnych programów) jest zadaniem prawie niemożliwym do wykonania. Znacznie łatwiej rozwiązać ten problem dokonując modularyzacji takiego wielkiego programu. Poszczególne moduły (podprogramy) mogą mieć np. po kilkadziesiąt linii – sprawdzenie poprawności takiego modułu jest znacznie łatwiejsze i niekiedy może zostać zautomatyzowane.

Po napisaniu podprogramu staje się on dostępny do wykorzystania – użycie go następuje po dokonaniu jego *wywołania.* Podprogramy często posiadają parametry (zwane również *argumentami*) – wartości tych parametrów są przekazywane w momencie wywołania podprogramu. Przykładem może być prosty podprogram arytmetyczny o nazwie Pierwiastek, który oblicza wartość pierwiastka kwadratowego z liczby, która została mu przekazana jako argument w momencie wywołania. Ogólnie podprogram ten definiowany jest jako:

```
Pierwiastek ( X )
```

Każde wywołanie zawierać natomiast powinno konkretną wartość, dla której pierwiastek ma zostać wyliczony (wartość ta staje się w danym wywołaniu wartością argumentu X ).

```
Pierwiastek ( 145 )          // pierwiastek z liczby 145
Pierwiastek ( alfa )         // pierwiastek z aktualnej wartości
                             // zmiennej alfa
```

Modularyzując program wyróżnia się program główny, od którego rozpoczyna się wykonanie całego programu i podprogramy, które są z programu głównego wywoływane. Oczywiście poszczególne podprogramy mogą się również nawzajem wywoływać.

Budując schemat blokowy podprogramu korzysta się z takich samych bloków jak przy tworzeniu programu głównego, jedynie w początkowym bloku START można podać nazwę podprogramu i jego argumenty. Podobnie w bloku STOP można określić wynik działania podprogramu.

W językach programowania określone są konstrukcje składniowe umożliwiające definiowanie i wywoływanie podprogramów, zwanych w tych językach raczej *funkcjami* lub *procedurami*. Ogólna zasada określa jako funkcje takie podprogramy, które przekazują pojedynczy wynik swojego działania (będący najczęściej liczbą). Przykładem funkcji jest więc omawiany podprogram Pierwiastek. Procedury nie mają pojedynczego wyniku, ich działanie może prowadzić do zmiany wartości wielu danych lub do wykonania różnych czynności organizacyjnych (np. wyświetlenia wyników na ekranie monitora). Sposób definiowania i wywoływania funkcji w języku C przedstawiono w rozdziale 7.

## 2.5. Przygotowanie i wykonanie programu

Do przygotowania i wykonania programu zazwyczaj stosowane są obecnie zintegrowane systemy programowania (IDE – *Integrated Development Environment*), które wspomagają programistę w niełatwym zadaniu napisania i sprawdzenia poprawności programu. Systemy te są niekiedy przeznaczone do programowania z użyciem tylko jednego języka programowania (np. języka C czy C++) albo umożliwiają wybór dowolnego z wielu udostępnianych języków. Przykładem tego ostatniego rodzaju systemu programowania jest system Microsoft Visual Studio.NET®. Ten właśnie system posłużył do przygotowania i przetestowania przykładowych fragmentów programów zawartych w tym podręczniku.

W fazie uczenia się programowania programista otrzymuje najczęściej jakieś zadanie z zakresu przetwarzania danych i przede wszystkim musi wymyślić algorytm rozwiązujący postawiony problem, a następnie zapisać ten algorytm jako program, wykorzystując wybrany język programowania. Wymyślony algorytm powinien uwzględniać możliwości stosowanego języka programowania. Jeżeli np. w wybranym języku nie jest dostępna operacja potęgowania liczb, to trzeba układając algorytm przewidzieć, w jaki sposób to potęgowanie będzie realizowane. Języki programowania, jak to już było omawiane, udostępniają zazwyczaj skromny zestaw struktur danych i jeszcze uboższy zestaw struktur sterowania. Tworząc algorytm rozwiązujący pewien problem przetwarzania danych należy więc brać pod uwagę możliwości języków programowania i kolejne kroki algorytmu formułować tak, aby było możliwe ich zapisanie w wybranym języku.

### 2.5.1. Edycja tekstu

Zapisywanie algorytmu, czyli pisanie programu w wybranym języku programowania polega na ręcznym wpisywaniu poszczególnych deklaracji i instrukcji (proces ten jest niekiedy wspomagany przez automatyczne generowanie fragmentów programu). Zintegrowane systemy programowania zawierają zazwyczaj specjalizowane edytory tekstowe, przystosowane właśnie do wprowadzania i modyfikowania tekstu programu (lub jak się niekiedy mówi *kodu programu* lub *kodu źródłowego*). Edytory te, poza standardowymi funkcjami dostępnymi w normalnych edytorach tekstu, oferują szereg udogodnień ułatwiających pisanie programu. Należą do nich między innymi:

- *kolorowanie składni*: edytor rozpoznaje niektóre standardowe fragmenty programu i nadaje im odrębny kolor, np. identyfikatory wyróżnione (if, while,...) – niebieskie, teksty – brązowe, itp.
- automatyczne stosowanie wcięć kolejnych linii programu, zależnie od stopnia ich zależności od linii poprzedniej, np.

```
if ( wart > 34 )                // język C
        ilo = 5 ;               // początek linii przesunięty w prawo (wcięcie)
```

- automatyczne generowanie fragmentów programów, np. szkieletów wybranych instrukcji:

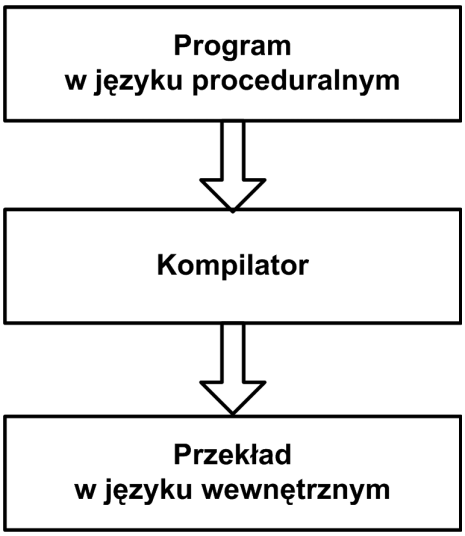
```
for (    ;    ;    )
{                }
```

- odszukiwanie nawiasu, odpowiadającego nawiasowi wskazanemu (bardzo użyteczne przy mocno zagnieżdżonych wyrażeniach czy instrukcjach).

Program źródłowy, przygotowany za pomocą edytora, jest wpisywany do pliku dyskowego o odpowiednim rozszerzeniu. Dla języka programowania C są to zazwyczaj rozszerzenia *.c* lub *.cpp*.

### 2.5.2. Kompilacja

Program zapisany w proceduralnym języku programowania ma zostać wykonany przez komputer. Ale, jak to przedstawiono w poprzednim rozdziale, procesor (czyli element komputera realizujący program) może wykonywać jedynie programy będące ciągami prostych poleceń, czyli programy zapisane w języku wewnętrznym. Należy więc tekst programu zapisany w języku proceduralnym przetłumaczyć na program w języku wewnętrznym, równoważny co do realizowanego algorytmu. Zadanie to wykonują specjalne programy (rys. 2.10) zwane *kompilatorami*.




Rys. 2.10. Kompilacja programu

Kompilator, opracowany dla danego proceduralnego języka programowania, traktuje tekst programu zapisany w tym języku (czyli program źródłowy) jako swoje dane wejściowe, odczytywane ze wskazanego pliku dyskowego. Kompilacja programu źródłowego składa się z następujących faz.

- Analiza leksykalna oddzielnych elementów programu źródłowego, czyli sprawdzenie, czy są one poprawnie zapisane (błędem leksykalnym jest np. zapisanie liczby całkowitej jako -34a12).
- Analiza składniowa deklaracji i instrukcji programu źródłowego, czyli sprawdzenie, czy elementy programu występują w sekwencji zgodnej z odpowiednią regułą gramatyczną – błąd składniowy występuje np. w następującym zapisie:

  ```
  for (beta < 1) else delta = 2 ;          // język C
  ```

  Poszczególne elementy tej linii są poprawne leksykalnie, ale sekwencja tych elementów jest niepoprawna.
- Analiza semantyczna, czyli sprawdzenie poprawności znaczenia poszczególnych wyrażeń programu źródłowego. Wykrywane są na tym etapie np. błędy wynikające z braku uprzedzających deklaracji identyfikatorów zmiennych czy funkcji, prób wykonywania operacji dla nieodpowiednich typów danych (np. dodawanie liczby całkowitej do tekstu) itp.
- Generacja przekładu – gdy wszystkie poprzednie analizy nie wykryją błędów w programie źródłowym, kompilator generuje przekład w języku wewnętrznym, czyli ciągi poleceń przeznaczonych do bezpośredniego wykonania przez procesor. Przekład ten jest zapisywany do pliku wykonywalnego.

Jeżeli użytkownik korzysta z systemu operacyjnego należącego do rodziny MS Windows®, to plik wykonywalny otrzyma rozszerzenie *.exe* i będzie miał strukturę wewnętrzną, odpowiednią dla tego typu plików.

Kompilator jest często składnikiem zintegrowanego systemu programowania i w związku z tym może być wykorzystywany do sprawdzania każdej wprowadzonej za pomocą edytora linii programu źródłowego. Pomoc kompilatora jest również niezbędna edytorowi do realizacji niektórych jego funkcji jak np. do odszukiwania nawiasu odpowiadającego nawiasowi zaznaczonemu.

### 2.5.3. Wykonanie programu

Po uzyskaniu pliku wykonywalnego, zawierającego przekład programu źródłowego, można przystąpić do jego wykonania. Jeżeli użytkownik korzysta ze zintegrowanego systemu programowania, to może uruchomić przekład za pomocą odpowiedniego polecenia tego systemu. W podręczniku niniejszym omawiane są wyłącznie programy korzystające z konsoli (punkt 1.2) – okno konsoli zostanie automatycznie otwarte przez system programowania, po wybraniu polecenia uruchomienia przekładu programu. Korzystając z tego okna można wprowadzać dane niezbędne wykonywanemu programowi i obserwować wyprowadzane wyniki.

Zintegrowane systemy programowania umożliwiają zazwyczaj wspomaganie procesu uruchamiania programu (ang. *debugging*). Poprawność leksykalna, składniowa i semantyczna programu, stwierdzona przez kompilator, nie świadczy niestety o jego pełnej poprawności, czyli nie gwarantuje, że przetłumaczony program jest poprawną realizacją wyjściowego algorytmu. Programując bardzo łatwo popełnić błąd, który w procesie kompilacji nie może zostać wykryty (kompilator nie otrzymuje żadnej informacji o algorytmie, który tłumaczony program ma realizować). Prostym przykładem takiej sytuacji jest zapisanie w programie instrukcji:

```
WartośćPartiiTowaru = LiczbaSztuk – CenaJednejSztuki ;                    // język C
```

Algorytm na pewno przewiduje dla obliczenia łącznej wartości partii towarów pomnożenie ich liczby przez cenę pojedynczej sztuki, użycie znaku odejmowania jest oczywistym błędem, niestety takiego błędu kompilator nie jest w stanie wykryć.

Kompilator, analizując program źródłowy i generując jego przekład może zapisać informacje dodatkowe, umożliwiające późniejsze wspomaganie uruchamiania tego programu. Korzystając z tych informacji system programowania pozwala na wykonywanie między innymi następujących operacji:

- zatrzymanie wykonywania programu we wskazanym jego miejscu (np. po osiągnięciu wskazanej linii),
- zatrzymanie wykonywania programu po wykonaniu zadanej liczby powtórzeń wskazanej pętli,
- zatrzymanie wykonywania programu, gdy spełniony zostanie zadany warunek (np. `Debet > 10000`),

- wznowienie wykonywania programu do końca lub do osiągnięcia następnego miejsca zatrzymania,
- wyświetlenie aktualnych wartości zmiennych i elementów struktur danych (wartości obliczonych do chwili zatrzymania wykonywania programu),
- zmiana wartości zmiennych i elementów struktur danych w miejscu zatrzymania – po wznowieniu wykonywania program będzie korzystał z nowo wprowadzonych wartości.

Wszystkie te operacje ułatwiają usunięcie błędów logicznych popełnionych podczas programowania. Gdy programista stwierdzi, że istnieje duże prawdopodobieństwo, iż jego program jest całkowicie poprawny, to można dokonać kompilacji tego programu raz jeszcze, uzyskując przekład docelowy, przeznaczony do normalnego użytkowania. Taka postać przekładu nie zawiera żadnych dodatkowych informacji wspomagających uruchamianie, dzięki czemu zajmuje mniejszy obszar pamięci i wykonuje się szybciej.

Przekład programu źródłowego może też być oczywiście wykonywany bez nadzoru systemu programowania. Należy w tym celu otworzyć okno konsoli (np. wybrać opcję systemu operacyjnego *Wiersz polecenia*), przejść do katalogu, w którym znajduje się plik z przekładem (plik o rozszerzeniu *.exe*) i wywołać program, wpisując jako polecenie nazwę tego pliku.

### 2.5.4. Interpretacja programu

W poprzednich dwu punktach przedstawiono sposób wykonania programu zapisanego w języku proceduralnym, polegający na jego przetłumaczeniu na postać wykonywalną (kompilacja) i wykonaniu tego przekładu. Nie jest to jedyny dostępny sposób wykonywania takich programów. Druga możliwość to posłużenie się *programem interpretującym* (*interpreterem*). Interpreter nie generuje przekładu, ale traktując program źródłowy jako swoje dane wejściowe, analizuje go i krok po kroku wykonuje zadane w nim operacje przetwarzania danych. Na przykład napotykając w programie źródłowym instrukcje:

```
Inwestycja = 1500 ;

Zysk = Inwestycja * 1.5 ;
```

interpreter zapisze w swoich strukturach wewnętrznych informację, że wartość zmiennej Inwestycja wynosi aktualnie 1500, a wartość zmiennej Zysk to 2250.

Korzystanie z programów interpretujących ma liczne wady, np. :

- analiza poprawności programu źródłowego jest mniej dokładna, niż w przypadku kompilacji – kompilator analizuje cały tekst programu źródłowego i dostarcza informacji o wszystkich znalezionych błędach, interpreter rozpoczyna wykonywanie programu źródłowego i grzęźnie z powodu prostych niekiedy błędów lub zbacza na trudne do przeanalizowania manowce,

- wykonanie programu źródłowego w trybie interpretacji trwa zdecydowanie dłużej, niż wykonanie przekładu tego programu,
- wielokrotne wykonanie programu źródłowego wymaga wielokrotnego uruchamiania interpretatora.

Jedyną zaletą interpreterów wydaje się być zaoszczędzenie czasu potrzebnego na kompilację programu źródłowego. Bilans wad i zalet interpretacyjnego wykonywania programów zapisanych w językach proceduralnych wskazuje na zdecydowana przewagę kompilacji i wykonywania przekładu. Toteż rzadko się już spotyka programy interpretujące przeznaczone dla profesjonalnych języków proceduralnych (np. dla języków C# czy Java). Interpretery pozostały jako narzędzie uruchamiania programów zapisanych w tzw. *językach skryptowych*. Języki te udostępniają zestaw standardowych poleceń (mogących posiadać parametry). Program zapisywany jest jako ciąg takich poleceń. Języki skryptowe stosowane są niekiedy do tworzenia aplikacji internetowych – przykładem może tu być pleniący się powszechnie język PHP.

## 2.6. Język programowania C

Język programowania C to *imperatywny*, proceduralny język programowania, który został zdefiniowany na początku lat 70-tych XX wieku. Jego autorem jest Denis Ritchie – pracujący w tym czasie w firmie Bell Labs. Język C bardzo szybko zdobył dużą popularność, między innymi dzięki temu, że był podstawowym językiem programowania dla nowego wówczas systemu operacyjnego Unix. Moduły programowe tego systemu operacyjnego zapisywane były w języku C (poza najbardziej wewnętrznymi modułami, które musiały być zapisane z użyciem języka wewnętrznego). Ponadto składnikiem systemu Unix był wbudowany kompilator języka C, umożliwiający tworzenie oprogramowania użytkowego. W krótkim czasie opracowano wiele kompilatorów języka C dla różnych typów komputerów. Autorzy tych kompilatorów interpretowali niekiedy definicję języka C na swój sposób i czasami wprowadzali do języka własne rozszerzenia. Skutkiem takich praktyk były coraz większe trudności z przenoszeniem programów zapisanych w języku C pomiędzy różnymi kompilatorami. Dla ujednolicenia definicji języka C i wprowadzenia jednej, powszechnie stosowanej wersji opublikowano w roku 1989 standard języka C, tzw. ANSI C. Po 10-ciu latach, w roku 1999, powstał następny standard tego języka, określany jako C99.

Język C stał się również punktem wyjścia do definiowania następnych generacji języków programowania. W latach 80-tych XX wieku opracowano definicję języka programowania C++, który jest rozszerzeniem języka C o mechanizmy programowania obiektowego. Dokonując tego rozszerzenia wprowadzono również dodatkowe konstrukcje do części nieobiektowej języka C++. W końcowej dekadzie tamtego wieku język C++ był najpowszechniej stosowanym na świecie językiem programowania. Kolejnym krokiem było wprowadzenie w roku 2002 obiektowego

języka programowania C#, który również wywodzi się z języków C i C++, ale jest językiem nieco wyższego poziomu (nie udostępnia mechanizmów, które być może przyspieszają wykonanie programu lecz umożliwiają równocześnie wprowadzanie do programu wielu trudnych do wykrycia błędów). Jak widać te trzy języki programowania powiązane są literą C, występującą w ich nazwie. Nie jest to tylko ukłon w stronę szacownych przodków – wiele definicji składniowych wprowadzonych w języku C bez zmian przeniesiono do języków C++ i C#. Ponadto język C stał się podstawą wielu specjalizowanych języków programowania (np. języków do zapisu algorytmów sterowania, języków wspomagających dla pakietów matematycznych itp.).

Twórca kompilatora danego języka programowania decyduje ostatecznie o tym, jakie elementy leksykalne i jakie konstrukcje składniowe mogą występować w programach źródłowych. Tworząc kompilator podejmuje się również decyzję o sposobie generowania przekładu instrukcji języka programowania, występujących w programie źródłowym, na sekwencje poleceń języka wewnętrznego. Czyli ostateczną definicję leksykalną, składniową i semantyczną języka programowania ustala autor kompilatora, przeznaczonego dla danego języka programowania. Przestrzeganie wspomnianych standardów języka C powinno doprowadzić do sytuacji, w której program uznany za poprawny przez jeden z dostępnych kompilatorów będzie również poprawny po przeanalizowaniu przez dowolny, inny kompilator tego języka. Rzeczywistość nie jest jednak tak idealna – pomimo istnienia obowiązujących standardów twórcy kompilatorów języka C często ulegają pokusie pomijania niektórych fragmentów standardu i dodawania własnych rozwiązań niestandardowych. Szczególnie często nie jest w pełni realizowany standard C99 – przyczyną jest być może fakt spadku zainteresowania językiem C, wypartym z powszechnego stosowania przez języki obiektowe.

Systemy programowania w języku C umożliwiają niekiedy wybór wersji języka C, która będzie akceptowana przez kompilator. Lista możliwości to na przykład: ANSI C, C99, EXT C (ten ostatni, to autorski pomysł twórców jednego z systemów programowania). Często można również, zadając odpowiednie opcje, definiować sposób interpretacji niektórych elementów języka. Możliwości te są szczególnie rozbudowane w kompilatorach z rodziny GNU C.

Powstaje więc pytanie, jaką wersję języka C przedstawić w dalszej części tego podręcznika. Język ANSI C nie jest już raczej stosowany, C99 nie zyskał szerszej popularności. Racjonalnym rozwiązaniem wydaje się w tej sytuacji ucieczka do przodu, polegająca na omówieniu nieobiektowej części języka C++. Ta nieobiektowa część języka C++ powstała przez dołączenie szeregu nowych elementów do języka ANSI C. Te dodatkowe możliwości bardzo się już upowszechniły i są szeroko stosowane w praktyce programistycznej. Taki wybór przedmiotu dalszego wykładu ułatwia również przyszłą naukę programowania w języku C++ czy C#.

Pozostałe rozdziały tego podręcznika są poświęcone omówieniu języka programowania C, rozumianego jako nieobiektowa część języka C++.

# 3. Dane

## 3.1. Alfabet

Do *alfabetu* języka C należą wszystkie znaki występujące w 8-bitowym kodzie ASCII. Wśród znaków tych wyróżnia się:

*litery* : A B . . . Z a b . . . z _

*cyfry* : 0 1 . . . 9

(do liter należy również znak podkreślenia). Znaki mające kody 32÷126 są wykorzystywane do zapisywania tekstu programu, znaki o kodach 0÷31 i kodach większych od 126 mogą występować jedynie jako elementy struktur danych lub łańcuchów znaków.

Nowsze kompilatory języka C stosują kodowanie UTF i dzięki temu umożliwiają stosowanie w tekście programu polskich liter diakrytyzowanych (ę, ą, ł, …).

We fragmentach programów występujących w tej książce są stosowane wspomniane polskie litery diakrytyzowane. Czytelnik, który chciałby przepisać program przykładowy do pliku dyskowego, celem dokonania jego kompilacji za pomocą kompilatora niekceptującego takich znaków, powinien zastąpić je odpowiednimi znakami łacińskimi.

W Dodatku A zamieszczono definicję kodu ASCII dla znaków, których kody mają wartości 0÷127. Dla wartości większych od 127 istnieje wiele wersji kodowania (tzw. strony kodowe). Szczegółowe informacje na ten temat można znaleźć w dokumentacji stosowanego systemu operacyjnego.

## 3.2. Identyfikatory

Ciągi znaków alfabetu języka C zaczynające się od litery i zawierające w dalszej części litery lub cyfry nazywane są *identyfikatorami*. Najczęściej kompilator analizuje pewną określoną liczbę początkowych znaków identyfikatora (np. 32 znaki), odróżniając duże i małe litery. Różnymi identyfikatorami są więc:

```
alfa   Alfa   AlfA   ALFA
```

Identyfikatory służą do nazywania tworzonych przez programistę elementów programu.

## 3.3. Słowa kluczowe

W języku C niektóre identyfikatory są zastrzeżone. Są one nazywane *słowami kluczowymi* i służą do oznaczania stałych składników programu. Nie wolno ich wobec tego używać jako nazw tworzonych przez programistę elementów programu. Edytory niektórych systemów programowania wyróżniają słowa kluczowe stosując wybrany kolor liter (np. niebieski). W przykładach zawartych w tej książce słowa kluczowe nie są wyróżninone żaden dodatkowy sposób. Lista słów kluczowych, znajdująca się w Dodatku C, obejmuje podstawowe słowa kluczowe języka C.

## 3.4. Komentarze

Tekst zawarty pomiędzy znakami /* i */ lub pomiędzy parą znaków // a końcem linii nie jest analizowany przez kompilator i służy komentowaniu programu. Kompilatory nie umożliwiają zazwyczaj zagnieżdżania komentarzy.

```
/* Definicja funkcji sortującej
   (ten tekst nie jest analizowany przez kompilator)
 */
//
// użyteczna uwaga
```

## 3.5. Typy danych liczbowych

Język C umożliwia przetwarzanie *liczb całkowitych* i *liczb zmiennopozycyjnych*. Istnieje pięć typów liczb całkowitych (*typy całkowitoliczbowe*):

```
char        short       int        long       long long
```

Liczby typu int oraz liczby typu long mają niekiedy taką samą reprezentację wewnętrzną i taki sam zakres wartości. Dla każdego typu liczb całkowitych można wybrać reprezentację uwzględniającą dodatni albo ujemny znak liczby (signed) lub znaku nie uwzględniającą (unsigned) Zakresy wartości poszczególnych typów liczb całkowitych zależą od zastosowanego kompilatora. Przykładowo w tabeli 3.1. przedstawione zostały zakresy wartości dla kompilatora Microsoft Visual Studio.NET®.

Tabela 3.1. Zakresy wartości typów całkowitoliczbowych

| typ | signed | unsigned | bajty |
|---|---|---|---|
| char | -128 , +127 | 0 , 255 | 1 |
| short | -32768 , +32767 | 0 , 65535 | 2 |
| int , long | -2147483648 , +2147483647 | 0 , 4294967295 | 4 |
| long long | – 9 223 372 036 854 775 808 ,<br>+ 9 223 372 036 854 775 807 | 0 ,<br>18 446 744 073<br>709 551 615 | 8 |

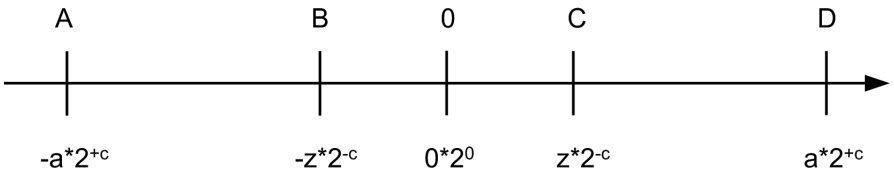
Dla liczb zmiennopozycyjnych istnieją w języku C trzy typy:

```
float    double    long double
```

Liczby zmiennopozycyjne są reprezentowane za pomocą *cechy* i *mantysy*.

$$lzmp = m * 2^c$$

Cecha *c* określa rząd wielkości liczby *lzmp*, a mantysa *m* zawiera początkowe cyfry tej liczby. Ten sposób pamiętania liczb zmiennopozycyjnych powoduje występowanie dwóch podzakresów reprezentacji (rys. 3.1.). Liczby rzeczywiste z zakresów B ÷ 0 i 0 ÷ C nie mogą być reprezentowane, w zakresach A ÷ B i C ÷ D dokładnie reprezentowane są tylko liczby rzeczywiste różniące się przynajmniej na pozycji najmłodszej cyfry przechowywanej w mantysie. W tabeli 3.2. przedstawiono zakresy wartości wymienionych trzech typów liczb zmiennopozycyjnych (znów dla kompilatora Microsoft Visual Studio.NET®)




Rys. 3.1. Liczby zmiennopozycyjne
a – maksymalna wartość mantysy, z – minimalna wartość mantysy,
c – maksymalna bezwzględna wartość cechy

Tabela 3.2. Zakresy wartości typów zmiennopozycyjnych

| typ | zakres | cyfry dokładne | bajty |
|---|---|---|---|
| float | $-3.4*10^{38}$ , $+3.4*10^{38}$ | 6 | 4 |
| double, long double | $-1.8*10^{308}$ , $+1.8*10^{308}$ | 15 | 8 |

## 3.6. Liczby

### 3.6.1. Liczby całkowite

W języku C *liczby całkowite* można zapisywać w notacji dziesiętnej, ósemkowej (oktalnej) lub szesnastkowej (heksdecymalnej).

```
12    154555                    // dziesiętnie
012   03777453                  // ósemkowo
0xAB    0x5c5d   0xfff45a       // szesnastkowo
```

Sposób reprezentowania liczby całkowitej zależy od jej wartości. Kompilator dobiera taką reprezentację, która ma najmniejszą liczbę bajtów wystarczającą do zapamiętania wprowadzanej wartości, rozpoczynając od reprezentacji typu int. Programista ma dodatkowo możność sterowania sposobem reprezentowania liczb całkowitych przez umieszczenie za ostatnią cyfrą przyrostków u,U,l,L. Przyrostki u lub U nakazują utworzenie reprezentacji typu unsigned (bez znaku ±), przyrostki l lub L powodują utworzenie reprezentacji typu long.

```
45211u                                  // unsigned int
15L     077777l     0xFF4FFFL           // signed long
3000000000ul        0xC56AFB44UL        // unsigned long
```

## 3.6.2. Liczby zmiennopozycyjne

*Liczby zmiennopozycyjne* mogą być zapisywane jako

*całość* . *ułamek* *całość* . *ułamek* E *wykładnik*

Składniki *całość* albo *ułamek* mogą być puste, składnik *wykładnik* może rozpoczynać się od znaku + albo – , a zamiast litery E można użyć litery e.

```
1.25    0.343   .5    2.  35.56E-12  0.34e2   5e3   17.18E+28
```

Liczby zmiennopozycyjne są standardowo reprezentowane jako dane typu double. Za pomocą przyrostków f, F, l, L programista może nakazać kompilatorowi utworzenie reprezentacji typu float (przyrostki f lub F) lub typu long double (przyrostki l lub L).

```
12.545f     0.2345676543F               // float
0.5e-3l     0.9999998899E456L           // long double
```

## 3.6.3. Znaki alfabetu

Pojedyncze *znaki alfabetu*, reprezentowane są w pamięci za pomocą liczb całkowitych typu char. Znaki te zapisuje się z użyciem apostrofów:

'*c*'

gdzie *c* może być znakiem widocznym (wprowadzanym bezpośrednio z klawiatury) lub opisem znaku. Opis znaku rozpoczyna się od lewego ukośnika, po którym następuje kod ASCII znaku, będący liczbą ósemkową lub szesnastkową.

```
'a'   '5'   '+'   '.'  'A'
'\071'   '\x41'   '\x5F'
```

Zamiast liczbowej wartości kodu można w opisach znaków stosować nazwy znaków sterujących, które podano w tabeli 3.3. np.:

```
'\n'   '\t'   '\a'
```

Tabela 3.3. Nazwy znaków sterujących

| opis znaku | kod znaku | znak |
|---|---|---|
| \a | 0x07 | BEL |
| \b | 0x08 | BS |
| \f | 0x0C | FF |
| \n | 0x0A | LF |
| \r | 0x0D | CR |
| \t | 0x09 | HT |
| \v | 0x0B | VT |
| \\ | 0x5C | \ |
| \' | 0x27 | ' |
| \" | 0x22 | " |
| \? | 0x3F | ? |

## 3.7. Łańcuchy znaków

Do zapisania ciągu znaków (czyli *łańcucha znaków* lub *tekstu*) stosowane są znaki cudzysłowu:

```
"Wprowadzenie do Programowania"
"Wynik : "
```

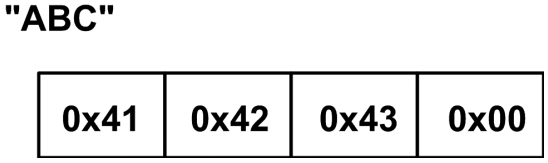
W łańcuchach mogą również występować opisy znaków:

```
"\tImię\tNazwisko\tMiejsce zamieszkania\n"
"\x16\x16\x02"              // SYN SYN STX
```

Jeżeli w łańcuchu ma wystąpić znak cudzysłowu ", to należy poprzedzić go znakiem lewego ukośnika:

```
"Spojrzał na mnie i powiedział: \"Nie mogę!\"."
```

Łańcuchy są pamiętane jako ciągi bajtów (tablice jednowymiarowe) zawierające kody ASCII kolejnych znaków łańcucha i zakończone bajtem o wartości zero.




Rys. 3.2. Reprezentacja łańcucha znaków

## 3.8. Zmienne

*Zmienna* to identyfikator związany z wartością, która podczas wykonywania programu może być modyfikowana. Każda zmienna, która jest używana w programie napisanym w języku C, musi zostać zadeklarowana przed jej użyciem. *Deklaracja zmiennej* ma postać:

*określenie_typu identyfikator_ 1*, *identyfikator_ 2*, ... , *identyfikator_ n* ;

*Określenie_typu* to słowo kluczowe (lub ciąg słów kluczowych) wskazujące, do jakiego typu danych należeć będzie wartość zmiennej. W zależności od użytego określenia mówi się o zmiennych całkowitych czy zmiennych zmiennopozycyjnych. W deklaracjach zmiennych mogą występować następujące określenia typu (określenia podane w tej samej linii są równoważne).

```
char              signed char
int               signed            signed int
short             short int         signed short int
long              signed long       long int          signed long int
unsigned char
unsigned          unsigned int
unsigned short    unsigned short int
unsigned long     unsigned long int
float
double
long double
```

Przykładowe deklaracje zmiennych:

```
int  i ;
char  a, b, c ;
unsigned long   przebyta_droga ;
double  masa, gęstość ;
```

Deklaracja zmiennej nie określa jej wartości, kompilator niekiedy nadaje deklarowanym zmiennym wartość zero. Zadawanie początkowej wartości zmiennych umożliwiają natomiast *definicje zmiennych*:

*określenie_typu identyfikator_1 = wartość_1,..., identyfikator_n = wartość_n ;*

gdzie *wartość_1, ... , wartość_n* są liczbami, znakami lub wyrażeniami arytmetycznymi. Na przykład :

```
int  LicznikWykonań = 125,  SumaWpłat = 0 ;
float  DokładnośćPomiaru = 0.0005,  UchybMiernika = 0.001 ;
double  MocTransformatora = 15e6,  StratyWŻelazie = 1500 ;
double   Promień,  Obwód,   DwaPi  = 6.28318530717959 ;
```

## 3.9. Stałe

*Stała* to identyfikator związany z wartością nie ulegającą zmianie podczas wykonywania programu. Stałe muszą być definiowane przed ich użyciem w programie, definicja stałej rozpoczyna się od słowa kluczowego const.

const *określenie_typu identyfikator_1 = wartość_1, .... ,*
*identyfikator_n = wartość_n* ;

Wartości stałej nie można w żaden sposób zmieniać, można ją tylko odczytywać (pobierać).

```
const int  DniTygodnia = 7,  TygodnieRoku = 52 ;
const float  Pi = 3.14159,  E = 2.71828 ;
const double  LiczbaAvogadro = 6.022E23 ;
```

## 3.10. Operator przypisania

Za pomocą operatora przypisania, którym w języku C jest znak równości = , dokonuje się zmiany wartości zmiennych. Po lewej stronie tego operatora mogą występować zmienne lub elementy struktur danych, a po prawej stronie wyrażenia.

```
int  LicznikObrotów,  SumaObrotów,  NowyTydzień;
//
LicznikObrotów  =  1 ;                  // zmienna LicznikObrótow przyjmuje wartość 1
//
SumaObrotów  =  LicznikObrotów ;        // zmienna SumaObrotów przyjmuje
                                        // wartość zmiennej LicznikObrótow
//
NowyTydzień  =  DniTygodnia ;           // zmienna NowyTydzień przyjmuje
                                        // wartość stałej DniTygodnia
```

## 3.11. Wskaźniki

Odrębną grupą typów danych języka C są *wskaźniki* określające adresy danych, struktur danych lub funkcji. Wskaźnik zawiera informację o lokalizacji wskazywanego elementu w pamięci komputera oraz informację o typie tego elementu (odróżnia się na przykład wskaźniki zmiennych typu int od wskaźników zmiennych typu long). Możliwe jest również definiowanie wskaźników zmiennych i struktur danych dowolnego typu, oznaczanego słowem kluczowym void (puste). Wskaźniki mogą być wartościami zmiennych wskaźnikowych, które deklaruje się poprzedzając identyfikator zmiennej znakiem gwiazdki * .

```
int  *Elementy,  *Składniki ;
long double  *Ułamek,  *BłądPomiaru ;
float  MocPoczątkowa,  MocPełna,  *Moc ;
/* zmienne typów liczbowych i zmienne wskaźnikowe tego samego  typu
   można deklarować łącznie */
void  *Dowolny,  *Każdy ;
```

Typem zmiennej wskaźnikowej Elementy jest int*, a typem zmiennej Dowolny jest void* (przy powoływaniu się na typ zmiennej wskaźnikowej gwiazdka jest dodawana do identyfikatora typu).

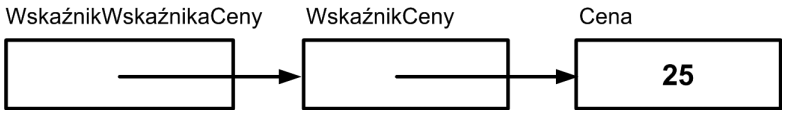
Tak zadeklarowane zmienne wskaźnikowe nie mają zadanych wartości początkowych, kompilator nadaje im niekiedy wartość zero (NULL). Wartością zmiennej wskaźnikowej danego typu może być wskaźnik innej zmiennej tego samego typu. Do wyznaczenia wskaźnika jakiejś zmiennej lub stałej używany jest *operator adresu* &. Za jego pomocą można nadawać wartości początkowe zmiennym wskaźnikowym i następnie w programie dokonywać zmian tych wartości.

```
int  Psy  =  5,  Koty  =  7 ;
int  *Zwierzę =  &Psy ;       /* wartością zmiennej Zwierzę staje się
                                 wskaźnik zmiennej Psy */
Zwierzę  =  &Koty ;           /* wartością zmiennej Zwierzę staje się
                                 wskaźnik zmiennej Koty */
float  Alfa,  Beta,  *PierwszaLitera,  *DrugaLitera ;
PierwszaLitera  =  &Alfa ;
DrugaLitera  =  &Beta ;
PierwszaLitera  =  DrugaLitera ;  /* wartością zmiennej PierwszaLitera
            staje się wskaźnik będący wartością zmiennej DrugaLitera */
//
float  JedenCal  =  2.54 ;
float  *Jednostka ;
long  *Długość ;
void  *Wartość ;
Jednostka  =  &JedenCal ;          // poprawnie
Długość  =  &JedenCal ;            // błąd
Wartość  =  &JedenCal ;            // poprawnie
```

Wartością zmiennej wskaźnikowej może również być wskaźnik innej zmiennej wskaźnikowej tego samego typu (w deklaracji pojawiają się wówczas dwa znaki gwiazdki **)

```
int  Cena  =  25,  *WskaźnikCeny,  **WskaźnikWskaźnikaCeny ;
WskaźnikCeny  =  &Cena ;
WskaźnikWskaźnikaCeny  =  &WskaźnikCeny ;
```

Po wykonaniu tego fragmentu programu pomiędzy zadeklarowanymi zmiennymi powstaną powiązania przedstawione na rysunku 3.3.




Rys. 3.3. Wskaźniki

*Wskaźniki łańcuchów znaków* mogą także być wartościami zmiennych wskaźnikowych. Łańcuch jest reprezentowany przez adres swojego pierwszego znaku.

```
/* początek programu */
char *Początek = "Początek obliczeń\n",
     *Koniec = "Koniec obliczeń\n", *Napis ;
Napis = Początek ;
/* wyświetlenie napisu i wykonanie obliczeń */
Napis = Koniec ;
/* wyświetlenie napisu i koniec programu */
```

Systemy programowania w języku C zawierają bogate biblioteki funkcji przetwarzających łańcuchy. Większość argumentów tych funkcji jest typu char*.

Odwołanie do zmiennej wskazywanej przez pewną zmienną wskaźnikową umożliwia *operator dostępu pośredniego* * . Operator ten zastosowany do zmiennej wskaźnikowej daje w wyniku wartość zmiennej wskazywanej.

```
int Długość = 5, Szerokość ;
int *Bok = &Długość;
Szerokość = *Bok ;                  // równoważne Szerokość = Długość ;
//
long Odległość, *Dystans = &Odległość ;
*Dystans = 685421 ;                 // równoważne Odległość = 685421 ;
//
float Napięcie = 1.5, Potencjał, *WskaźnikNapięcia = &Napięcie,
    **WskaźnikWskaźnikaNapięcia = &WskaźnikNapięcia ;
Potencjał = **WskaźnikWskaźnikaNapięcia ;
                                    // równoważne Potencjał = Napięcie ;
```

Wartością zmiennej wskaźnikowej nie może być wskaźnik stałej.

```
const double StawkaStała = 12e5 ;
double *WskaźnikStawki = &StawkaStała ;          // błąd
```

Zmiennej wskaźnikowej można nadać atrybut *wskaźnik do stałej* umieszczając w jej deklaracji, przed nazwą typu słowo kluczowe const, np.:

```
const long *NiePośredniczę ;
```

Wartościami zmiennej NiePośredniczę mogą być wskaźniki stałych typu long. Korzystając z dostępu pośredniego nie można jednak zmieniać wartości zmiennych wskazywanych.

```
long KwotaZaliczki ;
const long *UmówionaZaliczka = &KwotaZaliczki ;
KwotaZaliczki = 150L ;          // poprawnie
*UmówionaZaliczka = 150L ;      // błąd
```

*Stałą wskaźnikową* definiuje się natomiast przez umieszczenie słowa kluczowego const przed identyfikatorem. Stała wskaźnikowa musi mieć nadaną wartość początkową, która nie może być zmieniana podczas wykonywania programu.

```
float Nadwyżka = 12.5, Niedobór ;
float * const TylkoNadwyżka = &Nadwyżka ;
Nadwyżka = 15.8 ;               // poprawnie
*TylkoNadwyżka = 15.8 ;         // poprawnie
TylkoNadwyżka = &Niedobór;      // błąd
```

Jeśli deklaracja stałej wskaźnikowej rozpoczyna się również od słowa kluczowego const, to powstaje stała wskaźnikowa, której wartością jest wskaźnik do stałej.

```
const long double Giga = 1E9 ;
const long double * const Niezmiennik = &Giga ;
```

Identyfikator Niezmiennik nie może wystąpić po lewej stronie operatora przypisania.

## 3.12. Referencje

Następną grupą typów danych języka C są *referencje* zastępujące zmienne lub struktury danych. Każda operacja wykonywana na referencji jest identyczna z operacją wykonaną bezpośrednio na reprezentowanej przez tę referencję zmiennej czy strukturze. W definicji referencji jej identyfikator poprzedzany jest znakiem &, referencja musi mieć zawsze określoną wartość początkową, wiążącą referencję z reprezentowaną zmienną lub strukturą.

```
int LiczbaKrzeseł ;
int &Krzesła = LiczbaKrzeseł ;
```

Użycie referencji Krzesła powoduje od tego miejsca programu skutki identyczne użyciu zmiennej LiczbaKrzeseł.

```
long Podstawa, Fronton, &Fundament = Podstawa ;
Fundament = 12 ;                // równoważne Podstawa = 12 ;
Fronton = Fundament ;           // równoważne Fronton = Podstawa ;
//
float Powierzchnia, &Pole = Powierzchnia, *Obszar ;
Obszar = &Pole ;                // równoważne Obszar = &Powierzchnia ;
Obszar = Pole ;                 // błąd, podobnie jak Obszar = Powierzchnia ;
```

Typ wartości wpisywanej do referencji musi być taki sam jak typ zmiennej referencyjnej.

```
unsigned int WartośćBinarna = 0xCFCF ;
int &DwaBajty = WartośćBinarna ;                    // błąd
```

## 3.13. Typy wyliczeniowe

Definicja *typu wyliczeniowego* zawiera określenie zbioru stałych całkowitych, które mogą być wartościami zmiennych danego typu wyliczeniowego.

enum *identyfikator_typu* { *lista_stałych* } *lista_identyfikatorów_zmiennych* ;

*Identyfikator_typu* albo *lista_identyfikatorów_zmiennych* mogą w deklaracji typu wyliczeniowego nie wystąpić. Przy braku identyfikatora typu tworzone są jedynie zmienne definiowanego typu wyliczeniowego o identyfikatorach podanych na końcu jego definicji. *Lista_stałych* zawiera oddzielone przecinkami identyfikatory lub definicje stałych.

```
enum DniTygodnia { NIEDZIELA, PONIEDZIAŁEK, WTOREK,
                   ŚRODA, CZWARTEK, PIĄTEK, SOBOTA } ;
```

Taki zapis listy stałych nadaje stałej NIEDZIELA wartość 0, stałej PONIEDZIAŁEK wartość 1 itd. Aby niedziela była dniem tygodnia o numerze 1, należy w liście stałych użyć definicji stałej NIEDZIELA .

```
enum DniTygodnia { NIEDZIELA = 1, PONIEDZIAŁEK, WTOREK,
                   ŚRODA, CZWARTEK, PIĄTEK, SOBOTA } ;
```

Obecnie stała NIEDZIELA ma wartość 1, stała PONIEDZIAŁEK wartość 2 itd.

```
enum MowaKwiatow { GERBERA = 5, KONWALIA = 12, RÓŻA, MAK } ;
                   /* RÓŻA ← 13, MAK ← 14 */
enum UlubioneKolory { CZERWONY, POMARAŃCZOWY, RÓŻOWY = 0,
                      BRĄZOWY, ZIELONY, NIEBIESKI = 1, FIOLETOWY } ;
                    /* BRĄZOWY ← 1, ZIELONY ← 2, FIOLETOWY ← 2 */
```

Deklaracja zmiennej typu wyliczeniowego może wystąpić w definicji tego typu lub w innym miejscu programu, o ile typ wyliczeniowy ma swój identyfikator.

```
enum { A = 0x41, B, C, X = 0x58 } ZnakASCII ;
ZnakASCII = X ;          // poprawnie
ZnakASCII = 0x58 ;       // błąd
//
enum BOOL { FALSE, TRUE } ;
BOOL flaga, status = TRUE ;
flaga = FALSE ;
status = flaga ;
//
```

```
enum StanSilnika { ROZRUCH, PRACA, WYBIEG, STOP } StanAktualny ;
StanSilnika DmuchawaWstępna, *KolejnaDmuchawa ;
KolejnaDmuchawa = &DmuchawaWstępna ;
DmuchawaWstępna = WYBIEG ;
*KolejnaDmuchawa = ROZRUCH ;
```

## 3.14. Typ logiczny

W standardowym języku ANSI C nie występował odrębny typ logiczny złożony ze stałych reprezentujących *prawdę* i *fałsz*. Dla reprezentowania tych wartości stosowano następującą regułę:

- dowolna wartość liczbowa różna od zera reprezentuje prawdę,
- wartość zero reprezentuje fałsz.

Wprowadzenie odrębnego typu logicznego zwiększa czytelność programów i pozwala uniknąć trudnych do zauważenia błędów. Zmienne typu logicznego deklaruje i definiuje się za pomocą słowa kluczowego bool oraz stałych true i false.

```
bool Wykonane, KoniecObliczeń, Założenie = false ;
bool WLewo, WPrawo, *Zwrotnica = & WPrawo ;
Wykonane = Założenie ;
*Zwrotnica = true ;
```

Pomimo wprowadzenia typu logicznego nadal obowiązuje przedstawiona reguła reprezentowania prawdy i fałszu za pomocą wartości liczbowych. W wielu bibliotekach dołączanych do programu zapisanego w języku C znajduje się deklaracja typu wyliczeniowego BOOL pełniącego rolę typu logiczego (punkt 3.13). Można wówczas stosować typy bool i BOOL pamiętając jednak, jak zdefiniowane są stałe logiczne każdego z tych typów (np. true i TRUE).

## 3.15. Nazywanie typów danych

W definicji typu wyliczeniowego można podać jego identyfikator i w dalszej części programu wykorzystać wprowadzony identyfikator, do deklarowania zmiennych tego typu. Za pomocą definicji rozpoczynających się od słowa kluczowego typedef można natomiast *nazywać typy danych*, czyli przypisywać identyfikatory dowolnym typom danych.

typedef *typ* *identyfikator*

*Identyfikatorowi* przypisany zostanie *typ*, który może być dowolnym typem języka C. W dalszej części programu użycie tego identyfikatora jest równoważne z użyciem przypisanego mu typu.

```
typedef char* String ;
String Pytanie, Odpowiedź, Komentarz = "Bez komentarzy." ;
```

```
typedef   int   Numer ;
Numer  NumerDomu;
int  Liczba  =  15 ;
NumerDomu  =  Liczba ;          // typ Numer  jest równoważny z typem int
```

Niedozwolone jest jednak łączenie identyfikatora reprezentującego typ danych z innymi słowami kluczowymi.

```
typedef  long long    DużaLiczba ;
unsigned DużaLiczba    JednostkaAstronomiczna ;        // błąd
```

# 4. Wyrażenia

## 4.1. Definicja wyrażenia

W języku C *wyrażenia* składają się z wyrażeń elementarnych, operatorów oraz nawiasów okrągłych. Wyrażenia elementarne to liczby, znaki, zmienne, stałe, elementy struktur danych lub wywołania funkcji. Argumentami jedno- dwu- lub trójargumentowych operatorów mogą być wyrażenia lub wyrażenia elementarne. Nawiasy okrągłe umożliwiają ustalanie kolejności wykonywania operatorów. W rozdziale 3. przedstawiono operator przypisania ( = ), operator dostępu pośredniego ( * ) i operator wyznaczania adresu ( & ).

## 4.2. Konwersja wartości

Przed omówieniem dalszych operatorów należy przedstawić zasady *konwersji wartości*. Aby było możliwe wykonanie operatora dwuargumentowego (poza operatorem przypisania) wyrażenia występujące po jego lewej i prawej stronie muszą być tego samego typu. Jeżeli tak nie jest, to podczas translacji programu dokonywana jest konwersja wartości jednego z tych wyrażeń na równoważną wartość takiego typu, jaki ma drugie wyrażenie. Najpierw wartości typów char, unsigned char, signed char, short, enum są zamieniane na wartości typu int, a wartości typu unsigned short na wartości typu unsigned int. Następnie porównywane są typy obydwu argumentów operatora i wartość typu mniej dokładnego (reprezentowana za pomocą mniejszej liczby bajtów) jest zamieniana na wartość typu bardziej dokładnego (*konwersja rozszerzająca*). Przyjmuje się przy tym, że typy zmiennopozycyjne są bardziej dokładne od typów całkowitoliczbowych, gdy wartości obydwu tych typów są reprezentowane za pomocą takiej samej liczby bajtów. Konwersja rozszerzająca typów całkowitoliczbowych polega na powieleniu bitu znaku dla typów signed lub wyzerowaniu starszej części reprezentacji dla typów unsigned. Konwersja wartości całkowitej na wartość zmiennopozycyjną jest dokonywana przez zastąpienie liczby całkowitej liczbą zmiennopozycyjną o maksymalnie zbliżonej wartości. Wspólny typ, do którego zostały sprowadzone argumenty operatora, jest również typem wartości uzyskiwanej po wykonaniu operatora.

Konwersja wartości zachodzi także, gdy zmienna występująca po lewej stronie operatora podstawienia jest innego typu niż wartość wyrażenia znajdującego się po prawej stronie tego operatora. W tym przypadku zawsze dokonuje się konwersji

wartości wyrażenia na taki typ, jaki ma zmienna. Konwersja ta może być konwersją rozszerzającą lub *konwersją zawężającą*. W tym ostatnim przypadku stosowane są następujące zasady:

- konwersja pomiędzy wartościami typów zmiennopozycyjnych polega na zaokrągleniu mantysy,
- konwersja wartości zmiennopozycyjnych na całkowite polega na odrzuceniu części ułamkowej i ewentualnie starszych bitów części całkowitej,
- konwersja bardziej dokładnej wartości całkowitej na wartość mniej dokładną polega na pominięciu starszych bajtów liczby.

Ta ostatnia zasada może powodować całkowite zniekształcenie wartości poddawanej konwersji.

```
char  JedenBajt ;
JedenBajt  =  258 ;          // wartość zmiennej JedenBajt będzie równa 2
```

Konwersja typu może zostać wykonana również na życzenie programisty, który zadaje ją za pomocą *operacji rzutowania*. W języku C dozwolone są dwie postacie tej operacji:

*typ* ( *wyrażenie* ) lub ( *typ* ) *wyrażenie*

Stosując operację rzutowania, można uniknąć niepotrzebnych konwersji dokonywanych podczas obliczania wartości wyrażeń.

```
int  Całość  =  27 ;
float  Ułamek  =  0.14 ;
Całość  = Całość  +  Ułamek;          /* konwersja 27 na 27.0, dodawanie zmiennopozycyjne,
                                         konwersja 28.4 na 28, przypisanie */
Całość  =  Całość  +  int ( Ułamek ) ;  /* konwersja 1.4 na 1, dodawanie całkowitoliczbowe,
                                         przypisanie */
```

Operacja rzutowania może zostać wykonana również dla wskaźników – dozwolony jest wówczas jedynie drugi z podanych sposobów zapisu tej operacji.

```
int  Liczba,  *Adres  =  &Liczba ;
float  *Wartość ;
Wartość  =  ( float * ) &Liczba ;
Wartość  =  ( float * ) Adres ;
```

## 4.3. Operatory arytmetyczne

Do *operatorów arytmetycznych* należą:

| | | | |
|---|---|---|---|
| * | mnożenie | + | dodawanie |
| / | dzielenie | – | odejmowanie lub zmiana znaku |
| % | dzielenie modulo | | |

Operatory + i − są operatorami jedno- lub dwuargumentowymi, pozostałe operatory są dwuargumentowe. Zastosowanie jednoargumentowego operatora + nie zmienia wartości jego argumentu. Operator dzielenia modulo % jest zdefiniowany jedynie dla argumentów o wartościach całkowitych, inne operatory arytmetyczne są zdefiniowane dla dowolnych wartości liczbowych.

```
int DowolnaLiczba, Nieparzystość ;
Nieparzystość = DowolnaLiczba % 2 ;
/* 0 → parzysta, 1 → nieparzysta */
```

Podczas obliczania wyniku operacji arytmetycznych może wystąpić nadmiar stało- lub zmiennopozycyjny. Nadmiar stałopozycyjny, który może powstać przy wykonywaniu operacji na liczbach całkowitych, nie jest najczęściej sygnalizowany i powoduje zniekształcenie wyników obliczeń. Natomiast wystąpienie nadmiaru zmiennopozycyjnego jest zazwyczaj wykrywane i powoduje przerwanie wykonywania programu.

```
int Moskity = 1723000000, Komary = 2158000000, Krwiopijcy ;
Krwiopijcy = Moskity + Komary ;              // nadmiar stałopozycyjny
//
float Długość = 0.5e35, Szerokość = 0.2e5, Powierzchnia ;
Powierzchnia = Długość * Szerokość ;      // nadmiar zmiennopozycyjny
```

Realizacja operacji dodawania i odejmowania liczb zmiennopozycyjnych wymaga przekształcenia obu składników, tak aby ich cechy były równe. Efektem tego jest uzależnienie wartości sumy kilku liczb zmiennopozycyjnych od kolejności sumowania.

```
29.5e-17 + 29.5e-17 + 1  →  1.000000000000001
1 + 29.5e-17 + 29.5e-17  →  1.000000000000000
```

## 4.4. Operatory zwiększania i zmniejszania

Charakterystyczne dla języka C są unarne *operatory zwiększania i zmniejszania* wartości liczbowych:

++ zwiększenie argumentu o 1
−− zmniejszenie argumentu o 1

Argumentem operatorów zwiększania i zmniejszania mogą być zmienne lub elementy struktur danych posiadające trwałą, modyfikowalną reprezentację w pamięci komputera (ang. *l-value* ). Stąd argumentem tych operatorów nie może być wyrażenie ujęte w nawiasy (poza prostymi wyrażeniami postaci (zmienna) ). Każdy z tych operatorów może wystąpić przed argumentem, jako *operator prefiksowy* (*przedrostkowy*):

```
++alfa      −−beta
```

lub za argumentem, jako operator postfiksowy (przyrostkowy):

```
alfa++      beta−−
```

Jeżeli operator zwiększania lub zmniejszania występuje ze swoim argumentem wewnątrz wyrażenia, to przy zapisie prefiksowym operacja jest wykonywana **przed** obliczeniem wartości wyrażenia, natomiast przy zapisie postfiksowym operacja jest wykonywana **po** obliczeniu wartości wyrażenia.

```
float  Licznik = 2.5,  Suma;
Licznik ++ ;                        // równoważne  Licznik = Licznik  +  1 ;
++ Licznik  ;                       // równoważne  Licznik = Licznik  +  1 ;
Licznik −− ;                        // równoważne  Licznik = Licznik  –  1 ;
−− Licznik ;                        // równoważne  Licznik = Licznik  –  1 ;
Suma  =  ++( 2 * Licznik ) ;        // błąd
//
int  Jabłka = 1,  Gruszki = 2,  Owoce ;
Owoce =  Gruszki++  +  Jabłka;      // Owoce ← 3,   Gruszki  ← 3
Gruszki = 2 ;
Owoce =  ++Gruszki  + Jabłka ;      // Owoce  ← 4,  Gruszki  ← 3
```

## 4.5. Operatory relacyjne

Po wykonaniu *operatora relacyjnego* otrzymuje się wartość true (*prawda*), gdy relacja zadana przez operator zachodzi pomiędzy argumentami, albo wartość false (*fałsz*), gdy relacja nie zachodzi. Operatory relacyjne to:

```
<    mniejsze niż
<=   mniejsze lub równe niż
==   równe
!=   różne
>=   większe lub równe niż
>    większe niż
```

Operatory relacyjne są zdefiniowane dla dowolnych argumentów liczbowych. Za ich pomocą można również porównywać wskaźniki.

```
bool  Mniejsze, Równe, NieRówne, WiększeRówne ;
float  Procent  =  12.3, Grupa  =  5 ;
Mniejsze  =  Grupa  <  Procent ;
Równe  =  Grupa  ==  Procent ;
NieRówne  =  Grupa  !=  Procent ;
WiększeRówne  =  Grupa  >=  Procent ;
double  OpornikPierwszy  =  1.8E3,  OpornikDrugi = 5.3E3 ;        // 1.8 kΩ, 5.3 kΩ
double  *OpornośćWejściowa  =  &OpornikPierwszy ,
        *OpornośćWyjściowa  =  &OpornikDrugi ;
```

```
bool  PorownanieOporności ;
PorownanieOporności  =
    * OpornośćWejsciowa  ==  * OpornośćWyjściowa ;          // fałsz
 OpornośćWejściowa  =  &OpornikDrugi ;
 PorownanieOporności  =
    * OpornośćWejściowa  ==  * OpornośćWyjściowa ;          // prawda
```

## 4.6. Operatory logiczne

*Operatory logiczne* umożliwiają wykonywanie operacji logicznych pomiędzy argumentami traktowanymi jako *prawda* i *fałsz*. Dowolna niezerowa wartość liczbowa lub znakowa oraz stała true są interpretowane jako *prawda*, natomiast wartość 0 oraz stała false są interpretowane jako *fałsz*. Istnieją trzy operatory logiczne:

| | |
|---|---|
| ! | negacja |
| && | koniunkcja (iloczyn logiczny) |
| \|\| | alternatywa (suma logiczna) |

Operator negacji ! jest jednoargumentowy, pozostałe dwa operatory logiczne są dwuargumentowe. Gdy lewy argument koniunkcji && ma wartość *fałsz*, to wartość prawego argumentu nie jest obliczana. Podobnie gdy lewy argument alternatywy || ma wartość *prawda* nie oblicza się wartości prawego argumentu.

```
short  LewaGranica,  PrawaGranica,  Pomiar ;
/* ..... */
bool  JestWZakresie  =  LewaGranica  <  Pomiar  &&
                                Pomiar  <  PrawaGranica ;
int  Rok  =  1410 ;
bool  CzyRokPrzestepny;
CzyRokPrzestepny  =  ! (Rok % 4)  &&  Rok % 100  ||  ! (Rok % 400) ;
```

## 4.7. Operatory bitowe

Argumentami *operatorów bitowych* mogą być jedynie wartości typów całkowitoliczbowych. Operatory bitowe przetwarzają bezpośrednio ciągi binarne, będące reprezentacjami tych wartości.

| | |
|---|---|
| ~ | negowanie wszystkich bitów ciągu binarnego |
| & | koniunkcja (iloczyn logiczny) dwu ciągów binarnych |
| \| | alternatywa (suma logiczna) dwu ciągów binarnych |
| ^ | różnica symetryczna (suma modulo 2) dwu ciągów binarnych |
| << | przesunięcie ciągu binarnego w lewo z uzupełnieniem zerami |
| >> | przesunięcie ciągu binarnego w prawo z uzupełnieniem zerami lub z powieleniem bitu znaku |

Tabela 4.1. Operacje bitowe

| a | b | a & b | a \| b | a ^ b |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Operator negowania jest operatorem jednoargumentowym, pozostałe operatory bitowe są dwuargumentowe. Dla operatorów koniunkcji, alternatywy i różnicy symetrycznej wartość i-tego bitu rezultatu ustalana jest przez wykonanie odpowiedniej operacji na i-tych bitach argumentów. Definicje tych trzech operacji przedstawiono w tabeli 4.1. Operatory << i >> powodują przesunięcie ciągu binarnego będącego lewym argumentem o liczbę pozycji zadaną wartością prawego argumentu. Przyjmując, że ciąg binarny ma postać:

$b_{31}, b_{30}, \ldots, b_0$
dla operatora << jednokrotne przesunięcie jest realizowane jako:
$b_i = b_{i-1}$ dla $31 \le i \le 1$, $b_0 = 0$
Podobnie dla operatora >> jednokrotne przesunięcie to:
$b_{31} = 0$ gdy lewy argument jest typu unsigned
$b_{31} = b_{31}$ gdy lewy argument jest typu signed
$b_i = b_{i+1}$ dla $30 \le i \le 0$

Ponieważ bit $b_{31}$ jest najstarszym, a bit $b_0$ najmłodszym bitem liczby całkowitej reprezentowanej przez ciąg binarny, jednokrotne przesunięcie w lewo odpowiada pomnożeniu tej liczby przez 2, a jednokrotne przesunięcie w prawo podzieleniu jej przez 2. Prawy argument operatorów przesunięć nie powinien być ujemny ani większy od liczby bitów przesuwanej liczby całkowitej. Gdy po przesunięciu w lewo ciągu binarnego jest on następnie interpretowany jako liczba całkowita, należy liczyć się z możliwością wystąpienia niesygnalizowanego nadmiaru stałopozycyjnego.

```
int Podstawa = 35, Przeciwna, Parzystość,
                    Ósemka, Czterokrotność ;
Przeciwna = ~ Podstawa +1 ;       // równoważne Przeciwna = – Podstawa ;
Parzystość = Podstawa & 1 ;
Czterokrotność = Podstawa << 2 ;
                                  // równoważne Czterokrotność = Podstawa * 4 ;
Ósemka = Podstawa >> 3 ;
                                  // równoważne Ósemka = Podstawa / 8 ;
int Cena = 2000000000, NowaCena ;
NowaCena = Cena << 1 ;            // nadmiar stałopozycyjny
char Litera = 'k', Cyfra = 8, Klucz = 7, Równy ;
Litera = Litera & 0x5F ;          // zamiana małych liter na duże
```

```
Cyfra = Cyfra | 0x30 ;          // zamiana wartości binarnej 0-9
                                // na kod znaku ASCII
Równy = Klucz ^ 5 ;             // Równy == 0 jedynie, gdy Klucz == 5
```

## 4.8. Złożone operatory przypisania

Wyrażenia o postaci:

X = X ⊗ Y ;

można za pomocą złożonych operatorów przypisania zapisać jako:

X ⊗= Y ;

Istnieje 10 złożonych operatorów przypisania:

```
*=  /=  %=  +=  –=  <<=  >>=  &=  ^=  |=
```

Operatory te zadają wykonanie operacji arytmetycznej lub operacji bitowej oraz operacji przypisania wartości zmiennej.

```
double Kurs, Zwyżka ;
Kurs += Zwyżka;     // Kurs = Kurs + Zwyżka ;
int Status, Maska ;
Status &= Maska ;   // Status = Status & Maska ;
```

Wartością wyrażenia utworzonego przez operator przypisania i jego argumenty jest wartość prawego argumentu. Operatory przypisania mają takie same właściwości jak inne operatory i mogą w związku z tym być stosowane w dowolnym miejscu wyrażenia.

```
int Piki, Kara, Kiery ;
Piki = (Kara = 5) + 1 ;         // równoważne Kara = 5 ; Piki = Kara + 1 ;
Piki = Kara = Kiery = 0 ;       // Piki = 0 ; Kara = 0 ; Kiery = 0 ;
```

Konstruując tego typu wyrażenie należy unikać wielokrotnej zmiany wartości tej samej zmiennej w ramach jednego wyrażenia. Nieprzestrzeganie tej reguły może prowadzić do niejednoznaczności.

## 4.9. Operator warunkowy

W języku C istnieje jeden operator trójargumentowy. Jest nim *operator warunkowy*:

*wyrażenie_1* ? *wyrażenie_2* : *wyrażenie_3*

Argumenty operatora warunkowego mogą być dowolnego typu liczbowego lub wskaźnikowego. Wykonanie operacji zadanej operatorem warunkowym przebiega w następujący sposób:

- obliczenie wartości *wyrażenia_1* ,
- gdy wartością *wyrażenia_1* jest *prawda*, to rezultatem operacji jest wartość *wyrażenia_2* (*wyrażenie_3* nie jest obliczane),
- gdy wartością *wyrażenia_1* jest *fałsz*, to rezultatem operacji jest wartość *wyrażenia_3* (*wyrażenie_2* nie jest obliczane).

```
float Pierwszy, Drugi, Maksimum ;
/* ... */
Maksimum = Pierwszy > Drugi ? Pierwszy : Drugi ;
```

## 4.10. Operator rozmiaru

Unarny operator sizeof umożliwia ustalenie liczby bajtów, jaką zajmuje reprezentacja wartości określonego typu. Argumentem tego operatora może być wyrażenie lub nazwa typu, a wynikiem jest wartość całkowita.

```
long Linia ;
double Przelicznik ;
int Rozmiar = sizeof Linia ;          // Rozmiar ← 4
Rozmiar = sizeof ( long ) ;           // Rozmiar ← 4
Rozmiar = sizeof Przelicznik ;        // Rozmiar ← 8
Rozmiar = sizeof (double) ;           // Rozmiar ← 8
```

## 4.11. Operator wyliczenia

Znak przecinka , będący operatorem wyliczenia, umożliwia konstruowanie ciągu wyrażeń, którego wartością jest wartość ostatniego elementu ciągu.

```
long Kilometry, Godziny, Predkość, Średnia;
Średnia = ( Kilometry = 59287030,
            Godziny = 3121300,
Predkość = Kilometry / Godziny ) ;
                        // Prędkość ← 84000, Średnia ← 84000
//
float Alfa, Beta, Sigma;
Sigma = ( Alfa = 5.3, Beta = 2.5, Beta++ ) ;
                        // Beta ← 3.5, Sigma ← 2.5
```

## 4.12. Priorytety operatorów

Obliczenie wartości wyrażenia następuje poprzez wykonanie operacji wskazanych przez zawarte w tym wyrażeniu operatory. Przekład wyrażenia generowany przez translator jest tak skonstruowany, aby proces ten rozpoczynał się od najbardziej zagnieżdżonego podwyrażenia elementarnego nie zawierającego już nawia-

sów okrągłych. W takim podwyrażeniu kolejność obliczenia wyznaczają priorytety i łączność operatorów (Dodatek D). Jeżeli w wyrażeniu:

A ⊗ B ⊗ C

operator ⊗ jest lewostronnie łączny, to jest ono obliczane w kolejności:

(( A ⊗ B ) ⊗ C )

Natomiast dla operatora prawostronnie łącznego wyrażenie jest obliczane w kolejności:

( A ⊗ ( B ⊗ C ))

Większość operatorów to operatory lewostronnie łączne. Operatorami prawostronnie łącznymi są między innymi operatory przypisania i operator warunkowy.

```
char Asy, Króle, Damy, Walety , Dziesiątki ;
/* ... */
Asy + Króle – Damy – Walety + Dziesiątki ;
/* operatory lewostronnie łączne czyli
( ( ( Asy + Króle) – Damy ) – Walety ) + Dziesiątki ; */
//
Asy ? Króle : Damy ? Walety : Dziesiątki ;
/* operator prawostronnie łączny czyli
Asy ? Króle : ( Damy ? Walety : Dziesiątki ) ; */
```

# 5. Instrukcje

## 5.1. Rodzaje instrukcji

*Instrukcja* zawiera definicję obliczenia i określenie sposobu wykonania tego obliczenia. Program jest ciągiem instrukcji wykonywanych kolejno od pierwszej do ostatniej, o ile nie używa się, łamiących zasadę programowania strukturalnego, instrukcji skoku. Przekładem instrukcji generowanym przez kompilator jest fragment programu zapisanego w jezyku wewnętrznym procesora, realizujący obliczenie zdefiniowane tą instrukcją.

*Instrukcja prosta* nakazuje bezwarunkowe wykonanie pojedynczego obliczenia. Z dowolnego ciągu instrukcji można utworzyć *instrukcję złożoną*. *Instrukcja warunkowa* i *instrukcja wyboru* pozwalają na podejmowanie decyzji o dalszym przebiegu obliczeń, *instrukcje pętli* umożliwiają wielokrotne wykonywanie pewnego fragmentu programu.

## 5.2. Instrukcja prosta

Instrukcje proste to:

- instrukcja pusta złożona z samego znaku średnika ;
- dowolne wyrażenie zakończone znakiem średnika ;

Jeżeli w instrukcji nie występuje operator przypisania (bezpośrednio lub pośrednio w wywoływanej funkcji), to wykonanie tej instrukcji nie powoduje żadnego efektu, nie ulega zmianie wartość żadnej zmiennej.

```
int  Opakowania,  Kartony  =  1 ;
;                                   // instrukcja pusta
Opakowania  =  Kartony  –  1 ;      // zmiana wartości zmiennej Opakowania
Kartony++ ;                         // zmiana wartości zmiennej Kartony
Opakowania  +  Kartony ;            // poprawnie, ale bez rezultatu
```

## 5.3. Instrukcja złożona

Dowolny ciąg instrukcji ograniczony nawiasami klamrowymi { i } jest *instrukcją złożoną*. Instrukcja złożona jest równoważna instrukcji prostej, tzn. może wystąpić we wszystkich tych miejscach programu, w których dozwolone jest występowanie instrukcji prostej.

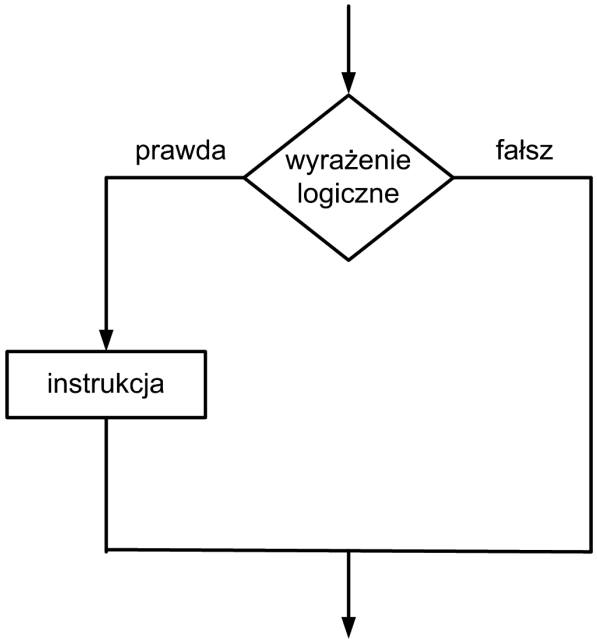
```
float Wymiar, Promień ;
Promień = 3.5 ;
Wymiar = 7.1 + 2 * Promień++ ;
//
{ Wymiar = 2 * Promień ; Promień = 1 ; }
/* średnik przed } nie może zostać opuszczony */
```

## 5.4. Instrukcja warunkowa

Podjęcie decyzji o wykonaniu lub pominięciu pewnego obliczenia możliwe jest za pomocą *instrukcji warunkowej* :

if ( *wyrażenie_logiczne* ) *instrukcja*

Gdy wartością *wyrażenia_logicznego* jest *prawda*, to wykonywana jest *instrukcja.* W przeciwnym przypadku (*fałsz*) instrukcja nie jest wykonywana (rys. 5.1.).

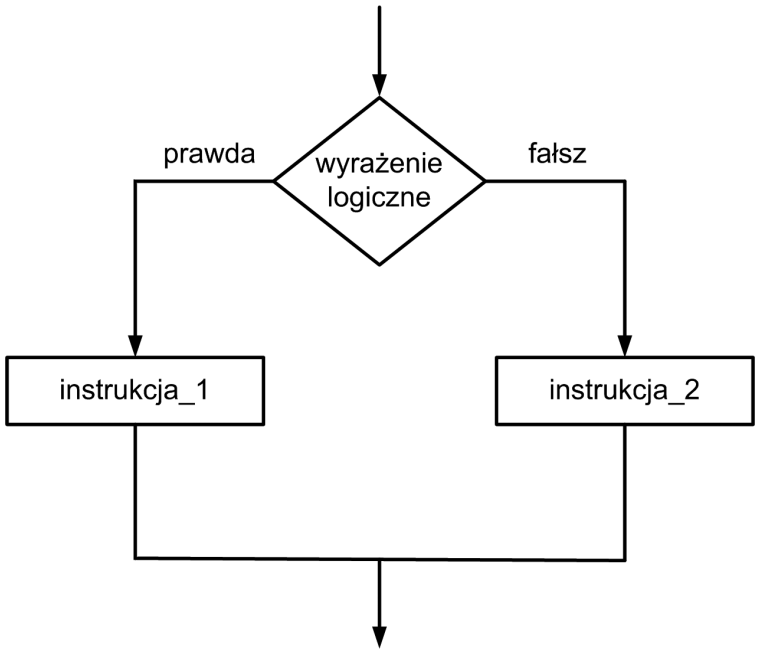



Rys. 5.1. Instrukcja warunkowa

```
long  Kolejny,  Wybrany ;
char  Flaga ;
/* ... */
if  ( Kolejny  > Wybrany )  Flaga  =  0 ;
//
if  ( Kolejny  < Wybrany )
{
Flaga  =  1 ;
Kolejny  = Wybrany  –  Kolejny ;
}
//
if  ( Kolejny  ==  Wybrany )
{
      if  ( Kolejny )
          Flaga  =  2 ;                    // Kolejny  !=  0
      if  ( ! Kolejny )
          Flaga  =  3 ;                    // Kolejny  == 0
}
```

Bardziej rozbudowana postać instrukcji warunkowej umożliwia dodatkowo określenie obliczenia, które zostanie wykonane, gdy wartość *wyrażenia_logicznego* będzie *fałszem* (rys. 5.2.)

if ( *wyrażenie_logiczne* ) *instrukcja_1* else *instrukcja_2*




Rys. 5.2. Rozszerzona instrukcja warunkowa

```
double Ma, Winien, Saldo, Debet ;
/* ... */
if ( Ma > Winien )
{
   Saldo = Ma – Winien ;
   Debet = –1 ;
}
else
{
   Saldo = –1 ;
   Debet = Winien – Ma ;
}
//
int Poziom, Granica ;
/* ... */
if ( Granica > Poziom ) Poziom = 3 ; else Poziom– – ;
/* średnik przed else nie może zostać opuszczony */
```

Obydwie postacie instrukcji warunkowej mogą być w sobie nawzajem dowolnie zagnieżdżane, co wymaga ustalenia zasady wiązania słów kluczowych if i else. W języku C obowiązuje zasada odnoszenia słowa else do najbliższego, niezwiązanego słowa if. A więc:

```
if ( A ) if ( B ) C ; else D ;
```

jest równoważne

```
if ( A ) { if ( B ) C ; else D ; }
```

Podobnie

```
if ( A ) if ( B ) C ; else if ( D ) E ; else F ;
```

odpowiada

```
if ( A ) { if ( B ) C ; else { if ( D ) E ; else F ; }}
```

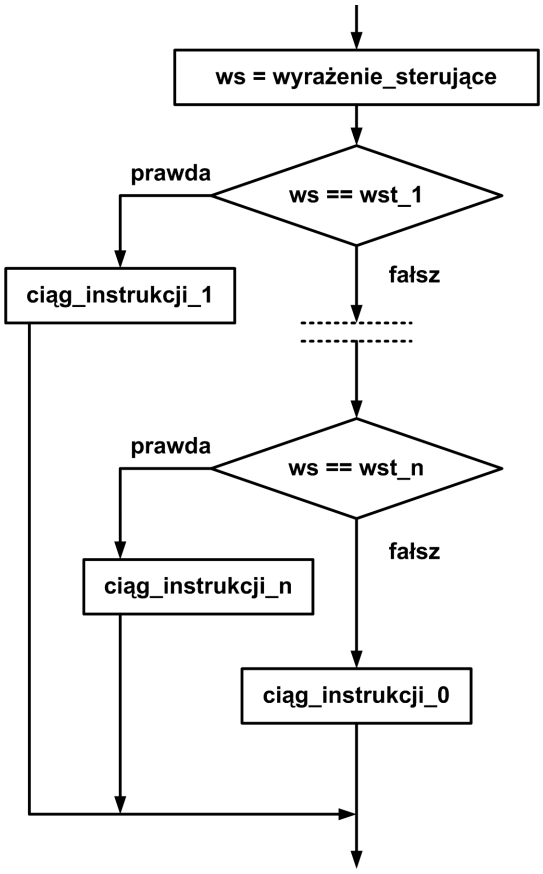
## 5.5. Instrukcja wyboru

*Instrukcja wyboru* pozwala na wykonanie jednego z wielu możliwych obliczeń, zastępuje ona wielokrotnie zagnieżdżoną instrukcję warunkową.

```
switch ( wyrażenie_sterujące )
   {
                case wyrażenie_stałe_1 : ciąg_instrukcji_1
                . . . . . .
                case wyrażenie_stałe_n : ciąg_instrukcji_n
                default : ciąg_instrukcji_0
   }
```

*Wyrażenia_stałe* to takie wyrażenia, których wartości można obliczyć podczas kompilacji. Realizacja instrukcji wyboru (rys. 5.3.) polega na obliczeniu wartości *wyrażenia_sterującego,* odszukaniu *wyrażenia_stałego* o takiej samej wartości i wykonaniu *ciągu_instrukcji* związanego z tym *wyrażeniem_stałym*. Każdy z *ciągów_instrukcji* związanych z poszczególnymi wartościami *wyrażenia_stałego* musi kończyć się instrukcją break powodującą zakończenie wykonywania instrukcji wyboru. Gdy poszukiwanie *wyrażenia_stałego* o wartości równej *wyrażeniu_sterującemu* nie zakończy się sukcesem, wykonywany jest *ciąg_instrukcji_0*. Fragment instrukcji wyboru rozpoczynający się słowem default może być pominięty – jeżeli w tym przypadku wartość *wyrażenia_sterującego* nie jest równa żadnemu *wyrażeniu_stałemu*, to nie wykonuje się żadnej instrukcji.

```
int LicznikA = 0, LicznikB = 0, LicznikXY = 0, Nieznany = 0 ;
char Znak ;
/* ... */
switch ( Znak )
{
   case 'A' : ++LicznikA ; break ;
   case 'B' : ++LicznikB ; break ;
   case 'X' :
   case 'Y' : ++LicznikXY ; break ;
   default  : ++Nieznany ;
}
//
int StanAktualny, StanNastępny ;
const int Maska = 0x3A ;
/* ... */
switch ( StanAktualny & Maska )
{
  case Maska & 0x02 : StanNastępny = 0x15 ;
                      StanAktualny = 0x21 ;
                       break ;
  case Maska & 0x30 : StanNastępny = 0x1F;
                      StanAktualny = 0x21;
                      break;
  default : StanAktualny = 0 ; StanNastępny = 0 ;
}
```



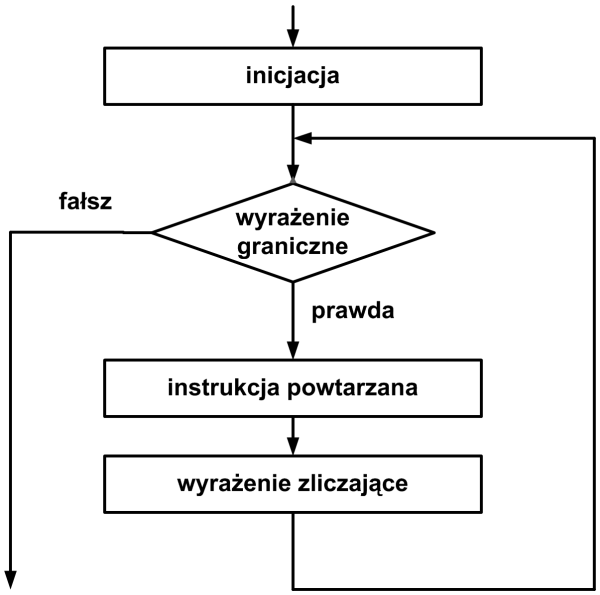

Rys. 5.3. Instrukcja wyboru

## 5.6. Instrukcje pętli

Wielokrotne wykonanie pewnego fragmentu programu można uzyskać za pomocą jednej z trzech istniejących w języku C *instrukcji pętli*. Instrukcja *pętli wyliczeniowej* ma postać :

for ( *inicjacja* ; *wyrażenie_graniczne* ; *wyrażenie_zliczające* )
*instrukcja_powtarzana*

Realizację tej instrukcji pętli (rys. 5.4.) rozpoczyna jednokrotne wykonanie *inicjacji* będącej definicją zmiennej licznikowej lub instrukcją ustalającą jej wartość (w *inicjacji* można również ustalić wartości kilku zmiennych licznikowych). Następnie oblicza się wartość *wyrażenia_granicznego* – jeżeli jego wartością jest prawda, to jest wykonywana *instrukcja_powtarzana*. Dalej oblicza się *wyrażenie_zliczające* i następuje powrót do obliczenia *wyrażenia_granicznego*. Opuszczenie pętli następuje po uzyskaniu przez *wyrażenie_graniczne* wartości reprezentującej *fałsz*.




Rys. 5.4. Instrukcja pętli wyliczeniowej

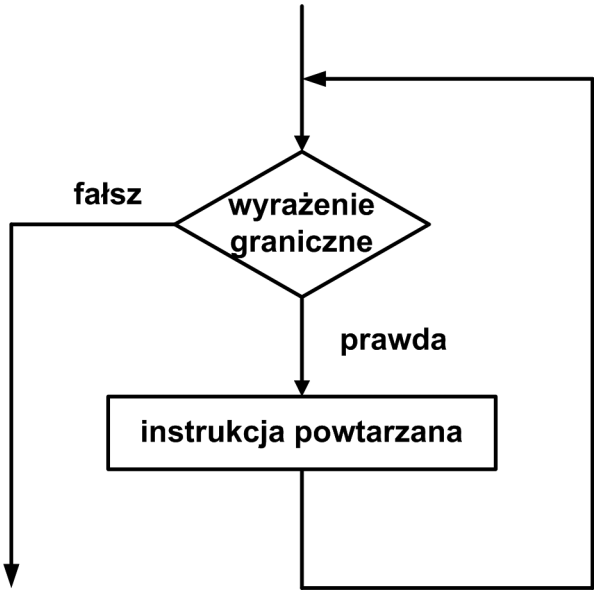
```
int  Suma  =  0, Kolejny ;
for  ( Kolejny  =  0 ;  Kolejny  <=  9 ;  Kolejny ++ )
/* inicjacja to instrukcja przypisania */
       Suma += Kolejny ;
//
char  *Napis  =  "aAb3Bc7C" ;
int  LiczbaWersalików  =  0 ;
for ( int  Pozycja  =  0 ;  Pozycja  <  8 ; Pozycja++ )
/* inicjacja to definicja zmiennej */
{
     if  ( *Napis  >=  'A'   &&  *Napis <= 'Z' )
              LiczbaWersalików++ ;
    else
                 if ( *Napis >= 'a'  && *Napis <= 'z' )
                          *Napis  =  *Napis  &  0x5F ;

     Napis++ ;
}
//  Napis ← "AAB3BC7C",   LiczbaWersalików ← 3
```

Instrukcja *pętli uprzedzającej* (lub *pętli dopóki*) ma postać :

while ( *wyrażenie_graniczne* ) *instrukcja_powtarzana*

Wykonanie tej pętli (rys. 5.5.) polega na obliczeniu wartości *wyrażenia_granicznego* – gdy reprezentuje ona *prawdę*, wykonuje się *instrukcję_powtarzaną* i powraca do obliczenia wartości *wyrażenia_granicznego*. Opuszczenie pętli następuje po uzyskaniu przez *wyrażenie_graniczne* wartości reprezentujacej *fałsz* .

```
char *Tekst = "pewien napis" ;
int Długość = 0 ;
while ( *Tekst++ ) Długość++ ;
 /* ostatni element łańcucha znaków ma wartość 0 (fałsz) */
//
float Suma = 1573.821, Składnik = 3.51;
int Licznik = 0 ;
while ( Suma > 1E-10 )
{
   Suma     –= Składnik ;
   Składnik *= Składnik ;
   Licznik++ ;
}
```

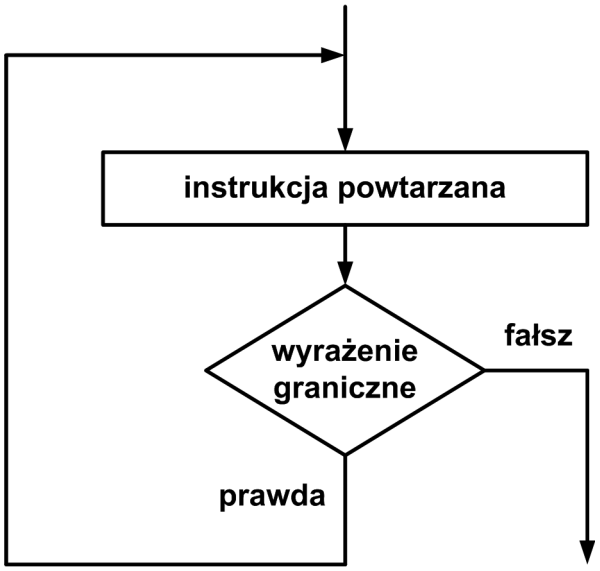



Rys. 5.5. Instrukcja pętli uprzedzającej

Dotychczas przedstawione instrukcje pętli najpierw dokonują sprawdzenia warunku wyjścia (czyli obliczenia wartości wyrażenia granicznego), a potem ewentualnie wykonują instrukcję powtarzaną. Jeżeli więc po pierwszym obliczeniu wartość wyrażenia granicznego okaże się fałszem, to instrukcja powtarzana w ogóle nie zostanie wykonana. Odwrotną właściwość ma instrukcja *pętli potwierdzającej* :

do *instrukcja_powtarzana* while ( *wyrażenie_graniczne* ) ;

Schemat realizacji tej pętli (rys. 5.6.) polega na wykonaniu *instrukcji_ powtarzanej*, po którym następuje obliczenie wartości *wyrażenia_granicznego*. Gdy wyrażenie to ma wartość reprezentującą *prawdę*, następuje powrót do wykonania *instrukcji_powtarzanej*, a w przeciwnym wypadku (*fałsz*) realizacja pętli kończy się.




Rys. 5.6. Instrukcja pętli potwierdzającej

```
long OdcinekAB = 3, OdcinekCD = 2 ;
do
{
   OdcinekAB *= OdcinekAB ;
   OdcinekCD += OdcinekCD ;
}
while ( OdcinekAB < OdcinekCD ) ;
/* OdcinekAB ← 9 , OdcinekCD ← 4 */
```

Niewłaściwa konstrukcja instrukcji pętli może doprowadzić do jej nieskończonego wykonywania. Przerwanie zapętlonego programu jest możliwie jedynie poprzez zewnętrzną interwencję operatora.

```
int Suma, Licznik ;
for ( int Kolejny = 0 ; Kolejny < 10 ; Licznik++ )
     Suma += Licznik ;                  // Kolejny nie zmienia się
//
float Wysokość = 3.485e2, Epsilon = 1.38534e-2 ;
long Powtórzenia ;
```

```
while  ( Wysokość  !=  0)
{
    Wysokość  −=  Epsilon ;
    Powtórzenia++ ;
}
/* Wysokosc  nigdy nie uzyska wartości dokładnie równej 0 */
//
unsigned char  Element = 5;
do
    Element  −=  2 ;
while  ( Element  !=  0 ) ;
/* Element nigdy nie będzie równe 0, Element ← 5, 3, 1, 255, 253, ... */
```

# 6. Struktury danych

## 6.1. Związki pomiędzy danymi

*Struktura danych* określa układ powiązań występujących w pewnym zestawie danych. W języku C istnieją dwa podstawowe rodzaje struktur danych: *tablice* i *struktury* (odmianą struktur są *unie*). Utworzone przez program egzemplarze struktur danych mogą być oznaczane identyfikatorami lub reprezentowane przez wskaźniki.

## 6.2. Tablice

*Tablica* to ciąg elementów tego samego rodzaju. Elementami tablicy mogą być dane (liczbowe lub wskaźnikowe) albo struktury danych. Jeżeli elementy tablicy ponumerowane są za pomocą pojedynczego indeksu (liczby całkowitej), to tablicę taką nazywa się tablicą *jednowymiarową*. Tablica jednowymiarowa jest reprezentacją wektora. Do numerowania elementów tablicy można też użyć par indeksów, wyróżniając *wiersze* i *kolumny*. Taka tablica to tablica *dwuwymiarowa,* będąca reprezentacją macierzy. Podobnie można tworzyć tablice o większej liczbie wymiarów.

### 6.2.1. Tablice jednowymiarowe

*Deklaracja tablicy jednowymiarowej* ma postać:

*typ identyfikator* [ *rozmiar* ] ;

gdzie *typ* może być dowolnym typem liczbowym, znakowym, logicznym, wskaźnikowym lub zdefiniowanym uprzednio typem struktury, a *rozmiar* jest wyrażeniem stałym, o wartości większej od zera, definiującym liczbę elementów tablicy (wartość wyrażenia stałego jest obliczana podczas kompilacji programu).

```
int Pomiary [ 5 ] ;
//
const int Sumy = 50, Różnice = 120 ;
float Składniki [ 2 * ( Sumy + Różnice ) ] ;
```

Elementy tablicy są numerowane za pomocą kolejnych liczb całkowitych, rozpoczynając zawsze od 0. Tablica Pomiary ma więc 5 elementów o numerach 0, 1, 2, 3, 4. Odwołanie do elementu tablicy zapisuje się, podając za identyfikatorem

tablicy zawarty w nawiasach kwadratowych [ ] indeks, będący wyrażeniem określającym numer tego elementu.

```
int  Pozycja ;
Pozycja  =  Pomiary [ 3 ] ;
//
Składniki  [Pozycja  +  2]  = Składniki [ Pozycja ]  +
                              Składniki [ Pozycja  +  1 ] ;
//
for  ( int  Kolejny  =  0 ;  Kolejny  <  5 ;  Kolejny++ )
     Pomiary [ Kolejny ]  =  48 ;
```

Deklaracja nie określa początkowej wartości elementów tablicy – kompilator nadaje niekiedy elementom deklarowanej tablicy wartość 0. Określenie początkowych wartości elementów tablicy umożliwia *definicja tablicy* :

*typ identyfikator* [ *rozmiar* ] = { *lista_wartości* } ;

*Lista_wartości* to oddzielone przecinkami wyrażenia stałe, których wartości są przypisywane kolejnym elementom tablicy:

```
long Kody [ 3 ]  =  { 154835,  337782,  0  } ;
/* Kody [ 0 ] ← 154835, Kody [ 1 ]  ← 337782, Kody [ 2 ] ← 0 */
```

Gdy w definicji podane są wartości wszystkich elementów tablicy, nie jest konieczne podawanie rozmiaru tablicy.

```
const  float  F1  = 3.5E7,  F2  =  33.1E8 ;
float  DaneF [  ]  =  {  F1 + F2,  F1 / ( F1 - F2 )  } ;
/* tablica DaneF ma 2 elementy */
```

Możliwe jest również definiowanie wartości tylko pewnej liczby początkowych elementów tablicy :

```
float  WspołczynnikiPQ [ 150 ]  =  { 1.5,  3.8,  3.8,  2.7 } ;
/* WspołczynnikiPQ [ 0 ] ← 1.5,   WspołczynnikiPQ [ 1 ] ← 3.8,
   WspołczynnikiPQ [ 2 ] ← 3.8,   WspołczynnikiPQ [ 3 ] ← 2.7,
   pozostałe  elementy ← 0  */
int Dane [ 1500 ] = { 0 } ;                    // zerowanie całej tablicy
```

Tablica znaków może być definiowana za pomocą łańcucha (tekstu). Należy przy tym pamiętać o zerze kończącym łańcuch, które również inicjuje kolejny element tablicy :

```
char  ZnakiTekstu [ 5 ]  =  "alfa" ;    //  ZnakiTekstu [ 4 ]  ← 0
char  LiteryTekstu [ 4 ]  =  "beta" ;   //  błąd, tablica zbyt mała
```

Elementami tablicy mogą być również wskaźniki :

```
int  Ołówki  =  1,  Długopisy  =  2,  Pisaki ;
int  *DoPisania [ 3 ]  = {  &Ołówki,  &Długopisy  } ;
```

```
DoPisania [ 2 ]  =  &Pisaki;
* DoPisania [ 2 ]  =  * DoPisania [ 0 ]  +  * DoPisania [ 1 ] ;
    //  Pisaki  ← 3
char  *Tydzień [  ]  =
          { "poniedziałek" , "wtorek" , "środa" ,
            "czwartek" , "piątek" , "sobota" , "niedziela" } ;
```

Z identyfikatorem tablicy kompilator wiąże wskaźnik początkowego elementu tablicy. Stąd zapisy

```
*Dane    i     Dane [ 0 ]
```

są równoważne.

```
char Bufor [ 8 ] ;
char  *Obszar ;
Obszar  =  Bufor ;
Obszar  = & Bufor [ 0 ] ;    /* przypisania równoważne */
```

Dalsze elementy tablicy można wskazywać, dodając numer elementu (czyli jego indeks) do wskaźnika elementu początkowego :

```
*( Dane + 4 )   jest równoważne   Dane [ 4 ]
```

Dodawanie to jest realizowane z uwzględnieniem rozmiaru pojedynczego elementu tablicy :

```
const  int  Rozmiar  =  25 ;
char   DaneZnakowe [ Rozmiar ] ;
long   DaneDługie [ Rozmiar ] ;
double  DaneNajwiększe [ Rozmiar ] ;
for  ( int Kolejny  =  0 ; Kolejny  <  Rozmiar  ;  ++Kolejny )
{
   *( DaneZnakowe  +  Kolejny )  =  1 ;                  // Kolejny * 1
   *( DaneDługie +  Kolejny )  =  1L ;                   // Kolejny * 4
   *( DaneNajwiększe  +  Kolejny ) = 1.0 ;               // Kolejny * 8
}
```

Za pomocą indeksów można również wskazywać kolejne pozycje łańcucha znaków:

```
char *CiągZnaków  =  "abbcaeb" ;
char Znak  =  'x' ;
/* znajdowanie pierwszej pary równych znaków ciągu */
for  ( int  Kolejny  =  0 ;  Kolejny  <  7 ;  ++Kolejny )
   if  ( CiągZnaków [ Kolejny ] == CiągZnaków [ Kolejny + 1 ]  )
   {
      Znak  =  CiągZnaków [ Kolejny ] ;
      Kolejny   =  7 ;                                   // zakończenie pętli
   }
```

Elementy tablicy są przez kompilator lokowane kolejno w jednym spójnym obszarze pamięci, którego rozmiar określony jest przez liczbę elementów tablicy i rozmiar pojedynczego elementu. Rozmiar obszaru zajętego przez tablicę można wyznaczyć za pomocą operatora sizeof zastosowanego do identyfikatora tablicy.

```
long Wpłaty [ 5 ] ;
int  Obszar, Element,  Elementów ;
Obszar  =  sizeof  Wpłaty;                                  //  20
Element  =  sizeof  Wpłaty [ 0 ] ;                          //  4
Elementów  =  sizeof  Wpłaty  / sizeof Wpłaty [ 0 ] ;       //  5
```

Dla tablic niedozwolone jest stosowanie operatora przypisania (kopiowanie).

```
int  Pierwsza [ 10 ],  Druga [ 10 ] ;
Druga = Pierwsza ;                          // błąd
```

## 6.2.2. Tablice wielowymiarowe

Tablica jednowymiarowa, będąca ciągiem elementów, jest reprezentacją wektora lub macierzy jednowierszowej (rozmiar tablicy jest liczbą kolumn). *Tablica dwuwymiarowa*, której elementami są ciągi takiej samej długości, jest reprezentacją macierzy. Definicja tablicy dwuwymiarowej ma postać :

*typ identyfikator* [ *liczba_wierszy* ] [ *liczba_kolumn* ] ;

Wiersze i kolumny są ponumerowane, począwszy od 0.

```
float Macierz [ 10 ] [ 20 ] ;     /* 10 wierszy, 20 kolumn */
```

Odwołanie do elementu tablicy dwuwymiarowej zapisuje się przez podanie dwu indeksów :

```
Macierz [ Wiersz ] [ Kolumna ]
```

gdzie *Wiersz* i *Kolumna* są wyrażeniami o wartościach nieujemnych, przy czym *Wiersz* jest numerem wiersza, a *Kolumna* numerem kolumny.

```
const int  Bok  =  30 ;
long Kwadrat [ Bok ] [ Bok ] ;
for  ( int  Wiersz  =  0;  Wiersz  <  Bok ;  ++Wiersz )
     for ( int  Kolumna  =  0 ;  Kolumna  <  Bok ; ++Kolumna )
       Kwadrat [ Wiersz ] [ Kolumna ] = Wiersz == Kolumna  ?  0  :  1 ;
   /* główna przekątna zostanie wyzerowana,   pozostałe elementy ← 1 */
```

Zadawanie wartości początkowych może dotyczyć wszystkich lub tylko niektórych elementów tablicy wielowymiarowej.

```
int  Pola [ 2 ] [ 3 ]  =  { { 1, 2, 3 } , { 4, 5, 6 } } ;
/* Pola [ 0 ] [ 0 ] ← 1, Pola [ 0 ] [ 1 ] ← 2,  Pola [ 0 ] [ 2 ]  ← 3
   Pola [ 1 ] [ 0 ] ← 4, Pola [ 1 ] [ 1 ] ← 5,  Pola [ 1 ] [ 2 ]  ← 6  */
```

```
float Lasy [ 3 ] [ 2 ] = {{ 8.5 }, { 3.2 }} ;
 /* Lasy [ 0 ] [ 0 ] ← 8.5, Lasy [ 0 ] [ 1 ] ← 0
    Lasy [ 1 ] [ 0 ] ← 3.2, Lasy [ 1 ] [ 1 ] ← 0
    Lasy [ 2 ] [ 0 ] ← 0,   Lasy [ 2 ] [ 1 ] ← 0  */
```

Jeżeli po identyfikatorze tablicy dwuwymiarowej podany zostanie tylko jeden indeks, to będzie on traktowany jako numer wiersza – wartością takiego odwołania jest wskaźnik zerowego elementu danego wiersza.

Utworzenie tablicy dwuwymiarowej, której wiersze są różnej długości, możliwe jest za pomocą tablicy wskaźników tablic.

```
int Wiersz0 [ 10 ], Wiersz1 [ 3 ], Wiersz2 [ 15 ] ;
int *Wiersze [ ] = { Wiersz0, Wiersz1, Wiersz2 } ;
**Wiersze = 0 ;                    // Wiersz0 [ 0 ]
**( Wiersze + 1 ) = 1 ;            // Wiersz1 [ 0 ]
*Wiersze [ 2 ] = 2 ;               // Wiersz2 [ 0 ]
* ( *Wiersze + 1) = 3 ;            // Wiersz0 [ 1 ]
* ( * ( Wiersze + 2) + 5) = 4 ;    // Wiersz2 [ 5 ]
```

Jak interpretowane jest odwołanie *( *( Wiersze + 2) + 5) dotyczące elementu Wiersz2 [ 5 ]? Najpierw wskaźnik związany z identyfikatorem Wiersze (wskazujący początkowy element tablicy Wiersze) zwiększa się, aby wskazywał na element tej tablicy o numerze 2. Dalej wykonuje się operację dostępu pośredniego wyznaczającą wskazywaną wartość – jest nią zgodnie z definicją tablicy Wiersze wskaźnik liczby całkowitej, a w tym przypadku wskaźnik początkowego elementu tablicy Wiersz2. Otrzymany wskaźnik jest znów zwiększany, by wskazywał na piąty element tablicy Wiersz2, a wykonywana jako ostatnia operacja dostępu pośredniego, umożliwia przypisanie temu elementowi nowej wartości.

Elementy tablicy dwuwymiarowej są umieszczane przez kompilator w spójnym obszarze pamięci. Kolejno lokowane są: elementy wiersza o numerze 0, następnie wiersza o numerze 1 itd. Operator sizeof umożliwia wyznaczenie rozmiaru całego obszaru zajętego przez tablicę dwuwymiarową, jak też obszaru zajętego przez jeden wiersz takiej tablicy.

```
long Lasy [ 2 ] [ 3 ] ;
sizeof Lasy ;              // 24
sizeof Lasy [ 0 ] ;        // 12
sizeof Lasy [ 0 ] [ 0 ] ;  // 4
```

Sposób deklarowania, definiowania i przetwarzania tablic dwuwymiarowych można rozszerzyć na większą liczbę wymiarów. Dla wymiaru i+1 kolejne rozmiary w definicji oraz indeksy w odwołaniu dodaje się przed rozmiarem czy indeksem wymiaru i.

```
float Trojaczki [ 5 ] [ 10 ] [ 20 ] ;
 /* tablica trójwymiarowa składająca się z 5 macierzy
    o 10 wierszach i 20 kolumnach każda */
```

```
for ( int Macierz = 0 ; Macierz <  5 ; ++Macierz )
    for ( int Wiersz = 0 ; Wiersz < 10 ; ++Wiersz )
        for (int Kolumna = 0 ; Kolumna  < 20 ; ++Kolumna )
                Trojaczki [ Macierz ] [ Wiersz ] [ Kolumna ] = 1.0F ;
```

Do przekładu programu nie jest dołączana procedura testowania poprawności indeksów. Odpowiedzialność za ograniczenie wartości indeksu tak, aby nie przekraczał on odpowiedniego rozmiaru tablicy, spoczywa na programiście. Indeksy o wartościach zbyt dużych w stosunku do rozmiaru tablicy powodują odczytywanie lub modyfikowanie zawartości przypadkowych bajtów pamięci, co może doprowadzić do zniszczenia programu.

## 6.3. Struktury

*Struktura* to zestaw pól, z których każde może być daną lub strukturą danych dowolnego typu. Pola te oznacza się identyfikatorami. Określając budowę struktury, definiuje się równocześnie nowy typ struktur, różny od dotychczas istniejących typów danych i typów struktur. Na podstawie tej definicji można następnie tworzyć egzemplarze struktury, które mają również swoje identyfikatory. W języku C definicję struktury można połączyć z zadeklarowaniem jednego lub kilku egzemplarzy tej struktury.

struct *typ_struktury* { *lista_pól* } *lista_identyfikatorów* ;

*Listę_pól* tworzą deklaracje kolejnych pól struktury będące deklaracjami zmiennych, tablic lub egzemplarzy struktur. *Lista_identyfikatorów* zawiera natomiast oddzielone przecinkami identyfikatory poszczególnych egzemplarzy struktury, które zostaną utworzone po zinterpretowaniu deklaracji struktury. *Typ_struktury* albo *lista_identyfikatorów* mogą w deklaracji struktury nie wystąpić:

- jeżeli nie występuje *typ_struktury*, to nie jest możliwe tworzenie dalszych egzemplarzy tego typu struktur,
- jeżeli nie występuje *lista_identyfikatorów*, to nie jest tworzony żaden egzemplarz struktury – utworzenie takich egzemplarzy jest możliwe w dalszej części programu za pomocą deklaracji:

  struct *typ_struktury* *lista_identyfikatorów* ;

W języku C słowo kluczowe struct rozpoczynające deklarację egzemplarzy struktury może być opuszczone (było ono niezbędne w języku ANSI C).

```
struct
{       int  Szerokość ;
        int  Dokładność ;
        char Konwersja ;
}  wzorzec ;
```

```
struct
{
    float  Re ;
    float  Im ;
}  z0, z1, z2 ;
//
struct  Osoba
{
    char  *Imię ;
    char  *Nazwisko ;
    int  RokUrodzenia ;
} ;
struct  Osoba  Ala,  Ola,  Ela ;
//
struct  Adres
{
    char  Miejscowość [ 16 ] ;
    char  Ulica [ 32 ] ;
    int  NumerDomu ;
    int  NumerMieszkania ;
} ;
Adres  AdresAli,  AdresOli,  AdresEli ;
```

Definicja struktury Osoba zawiera pola Imię i Nazwisko, w których są pamiętane wskaźniki ciągów znaków. Znaki składające się na imię i nazwisko muszą więc być przechowywane poza strukturą Osoba. Natomiast struktura Adres zawiera pola Miejscowość i Ulica będące tablicami znaków, do których bezpośrednio mogą być wpisywane znaki tworzące nazwy miejscowości i ulicy. Pierwsze z tych rozwiązań (struktura Osoba) charakteryzuje się większą oszczędnością pamięci (zajmowane jest tylko tyle bajtów, ile w konkretnym przypadku potrzeba dla zapamiętania imienia i nazwiska). Tak skonstruowane egzemplarze struktur są jednak trudniejsze w przetwarzaniu i kłopotliwe w przypadku zapisywania ich do plików dyskowych. Rozwiązanie drugie (struktura Adres) jest mniej oszczędne pod względem zajętości pamięci, ale bardziej wygodne przy przetwarzaniu egzemplarzy tej struktury i przy ich zapisie do plików dyskowych.

Jako pole struktury można zadeklarować inną strukturę. Deklaracja takiego pola zawiera definicję struktury (bez identyfikatora typu struktury) lub identyfikator struktury zadeklarowanej uprzednio.

```
struct  Silnik
{
      float  Pojemność ;
      char  *Paliwo ;
} ;
```

```
struct Samochod
{
   struct
   {
      char *Producent ;
      char *Model ;
   } marka ;
  Silnik napęd ;
} ;
```

Po zadeklarowaniu pewnej struktury możliwe jest również deklarowanie zmiennych, których wartościami są wskaźniki tego typu struktur.

```
struct DaneWejściowe
{
    int Poziom ;
    int Zakres ;
} Początek ;
struct DanePośrednie
{
   float Średnia ;
   long Suma ;
} Etap ;
DaneWejściowe *Start ;
DanePośrednie *Obliczenie ;
Start = &Początek ;
Obliczenie = &Etap ;
Start = &Etap ;                    // błąd, nieodpowiedni typ struktury
Obliczenie = Start ;               // błąd, nieodpowiedni typ wskaźnika
```

Deklarowanie egzemplarzy struktury można połączyć z nadaniem wartości początkowych poszczególnym jej polom.

```
struct Litery
{
   char Pierwsza ;
   char Ostatnia ;
   int Liczba ;
} seria1 = { 'a', 'b', 37 } , seria2 = { 'x', 'y', 35 } ;
Litery seria3 = { 'k', 'm', 36 } ;
```

Dostęp do pól wybranego egzemplarza struktury można uzyskać korzystając z identyfikatora lub ze wskaźnika. W pierwszym z tych przypadków stosuje się operator będący kropką, w drugim operator o postaci strzałki, utworzonej ze znaków – i > .

```
struct Podpis
{  char *Imię ;
   char  Inicjał ;
   char *Nazwisko ; } ;
Podpis DyrektorTechniczny, *Uprawniony = &DyrektorTechniczny ;
DyrektorTechniczny . Inicjał = 'A' ;
Uprawniony –> Inicjał = 'K' ;
//
if ( DyrektorTechniczny . Inicjał > 'Z' )
Uprawniony –> Inicjał = DyrektorTechniczny . Inicjał & 0x5F ;
```

Dla egzemplarzy struktur tego samego typu dozwolone jest stosowanie operatora przypisania.

```
Podpis AK = { "Anna", 'M', "Kowalska" }, NN ;
NN = AK ;
/* równoważne NN . Imię = AK . Imię ;
              NN . Inicjał = AK . Inicjał ;
              NN . Nazwisko = AK . Nazwisko ;  */
```

Egzemplarze struktur mogą być również elementami tablic. Tablicę taką tworzy się, podając po identyfikatorze struktury rozmiar tablicy.

```
struct Kartoteka
{
   char Nazwa [ 16 ] ;
   int Liczba ;
} komputery [ 100 ] ;
Kartoteka drukarki [ 100 ] ;
/* ..... */
int LiczbaKomputerów = 0, LiczbaDrukarek = 0 ;
for ( int Kolejny = 0 ; Kolejny < 100 ; ++Kolejny )
{
    LiczbaKomputerów += komputery [ Kolejny ] . Liczba ;
    LiczbaDrukarek += drukarki [ Kolejny ] . Liczba ;
}
```

Ważną rolę odgrywa możliwość rekurencyjnego definiowania struktur – pole struktury może zawierać wskaźnik do struktury właśnie definiowanego typu. Właściwość ta pozwala na tworzenie jedno- lub dwukierunkowych list oraz na budowanie reprezentacji grafów.

```
const int NULL = 0;
struct Lista
{
     char  *Etykieta ;
     Lista *Poprzedni ;
     Lista *Następny ;
}   Element0  =  { "AAA",  NULL, NULL  } ;
Lista  Element1  =  { "BBB" } ,  Element2  =  { "CCC" } ;
/* utworzenie dwukierunkowej listy cyklicznej */
Element0 . Następny  = &Element1 ;
Element1 . Następny  = &Element2 ;
Element2 . Następny  = &Element0 ;
Element0 . Poprzedni = &Element2 ;
Element1 . Poprzedni = &Element0 ;
Element2 . Poprzedni = &Element1 ;
```

Operator sizeof zastosowany do identyfikatora egzemplarza struktury lub do identyfikatora typu struktury umożliwia ustalenie wielkości obszaru pamięci zajmowanego przez tę strukturę.

```
struct  Zakres
{
    float  Początek ;
    float  Koniec ;
}  przedział ;
sizeof  przedział ;              //  8
sizeof  ( Zakres ) ;             //  8
```

Do deklarowania struktur można również użyć polecenia typedef. Wprowadzony za pomocą tego polecenia identyfikator typu struktury można następnie wykorzystywać do deklarowania kolejnych egzemplarzy takich struktur – w deklaracjach tych nie występuje ponownie słowo kluczowe struct .

```
typedef struct
{
     float  Liczba ;
     float  Dokladność ;
}  Wynik ;
Wynik  wynik1,  wynik2,  wynik3 ;
```

## 6.4. Unie

Kolejne pola egzemplarza struktury są przez kompilator lokowane w kolejnych bajtach pamięci. Unia jest natomiast strukturą, której pola są lokowane w tym samym wspólnym obszarze pamięci. Wielkość tego obszaru jest tak dobrana, aby po-

mieściła się w nim dana o reprezentacji zajmującej największą liczbę bajtów (spośród danych zadeklarowanych jako pola unii).

```
union
{
    char  Znak ;
    int    LiczbaCałkowita ;
    float  LiczbaDługa ;
}  trójka ;
```

Egzemplarz unii o identyfikatorze trójka zajmie 4 bajty konieczne dla pomieszczenia danej typu float. Taka organizacja unii powoduje, że spośród wszystkich pól sensowną wartość ma jedynie to pole, dla którego ostatnio wykonano operację przypisania. Wartości pozostałych pól są przypadkowe i nie należy ich odczytywać.

```
trójka . Znak  =  'A' ;                  // LiczbaCałkowita, LiczbaDługa niezdefiniowane
trójka . LiczbaCałkowita = 1357 ;    // Znak, LiczbaDługa niezdefiniowane
trójka . LiczbaDługa = 154.328;      // Znak, LiczbaCałkowita niezdefiniowane
```

Stosowanie unii pozwala na zaoszczędzenie obszarów pamięci, które w strukturach rezerwowane byłyby dla wykluczających się wariantów.

```
struct  PublikacjaNaukowa
{
    char  Autor [ 64 ] ;
    char  Tytuł [ 64 ] ;
    int  Rok ;
    union
    {
       struct
       {
            char  TytulCzasopisma [ 64 ] ;
            short  Numer ;
            short  Strona ;
       }  Czasopismo ;
       struct
       {
            char  NazwaKonferencji [ 64 ] ;
       }  MateriałyKonferencyjne ;
    }  MiejscePublikacji ;
     } ;
int Obszar = sizeof   PublikacjaNaukowa  ;        //  200
```

Dzięki takiej deklaracji w egzemplarzach struktur typu PublikacjaNaukowa rezerwuje się jedynie 68 bajtów przeznaczanych zamiennie albo na opis czasopisma, albo na nazwę konferencji.

# 7. Funkcje

## 7.1. Program główny i podprogramy

Podział programu na podprogramy i program główny, z którego podprogramy są wywoływane, jest jedną z podstawowych metod stosowanych w programowaniu. Umożliwia ona modularyzację programu i pozwala na uniknięcie wielokrotnego wpisywania ciągu takich samych instrukcji. W języku C podprogramy tworzy się za pomocą *funkcji*, które zastosowane do argumentów podanych w wywołaniu obliczają wartość (wynik) określonego typu. *Definicja funkcji* wiąże identyfikator funkcji, typ jej wyniku, listę argumentów i ciąg instrukcji realizowany po wywołaniu funkcji. Możliwe jest również definiowanie funkcji nie posiadających wyniku, będących odpowiednikami procedur znanych z innych języków programowania. W tym przypadku efektem działania funkcji są jedynie jej skutki uboczne (np. wyświetlenie wyników obliczeń na ekranie monitora).

Definicje funkcji nie mogą być w języku C zagnieżdżane, czyli np. definicja funkcji FuJeden nie może być umieszczona w definicji funkcji FuDwa. Generalną zasadą obowiązującą w języku C jest natomiast zakaz używania identyfikatora, który nie został jeszcze zadeklarowany. Specjalnego rozwiązania wymaga więc problem zapisania wzajemnie wywołujących się funkcji (w definicji funkcji FuJeden występuje wywołanie funkcji FuDwa, a w definicji funkcji FuDwa wywołanie funkcji FuJeden ). Problem ten w języku C rozwiązano przez wprowadzenie *deklaracji funkcji* (zwanych również *zapowiedziami funkcji*), które wiążą listę argumentów funkcji i typ jej wyniku z identyfikatorem funkcji.

## 7.2. Deklaracja funkcji

*Deklaracja funkcji* ma postać:

*typ identyfikator_funkcji* ( *lista_argumentów_formalnych* ) ;

gdzie:

*typ* określa typ wyniku funkcji: może to być dowolny typ liczbowy, wskaźnikowy lub referencyjny,

*lista_argumentów_formalnych* zawiera oddzielone przecinkami deklaracje argumentów o postaci:
*typ identyfikator_argumentu_formalnego*

*Argumenty formalne* mogą być dowolnego typu, niedozwolone jest grupowanie w jednej deklaracji kilku argumentów tego samego typu.

```
int PoleProwierzchni ( int Długość, int Szerokość ) ;
float WartośćTowaru ( float Cena, int Liczba ) ;
char* Odpowiedź ( char* Pytanie ) ;
double KątAlfa ( double BokA, BokB) ;       // błąd
```

Jeżeli typ wartości funkcji nie zostanie podany, to większość kompilatorów sygnalizuje błąd składniowy (starsze kompilatory domyślnie przyjmowały typ int). Funkcja, która nie wyznacza swojej wartości (wyniku) jest deklarowana jako funkcja o typie void (pusty). Liczba i rodzaj argumentów funkcji to *sygnatura funkcji*.

```
Max ( int Pierwszy, int Drugi ) ;           // błąd
void Druk ( char *Napis ) ;                 // bez wyniku
```

## 7.3. Definicja funkcji

*Definicja funkcji* wiąże typ wartości funkcji, identyfikator i sygnaturę funkcji z ciągiem deklaracji i instrukcji definiującym podprogram obliczający wartość funkcji. Podprogram ten może powodować efekty uboczne, związane ze zmianą wartości zmiennych lub elementów struktur danych czy też dotyczące urządzeń zewnętrznych.

*typ identyfikator_funkcji* ( *lista_argumentów_formalnych* ) { *blok* }

gdzie *blok* jest ciągiem złożonym z deklaracji lub definicji danych i struktur danych oraz z instrukcji. Wykonywanie bloku kończy się dla funkcji obliczających wartość po napotkaniu instrukcji zakończenia o postaci:

return *wyrażenie* ;

obliczona wartość *wyrażenia* staje się wartością (wynikiem) funkcji. Dla funkcji, które nie obliczają wartości (typ void ) wykonanie bloku kończy instrukcja o postaci:

```
return ;
```

Instrukcję return kończącą blok można opuścić, może ona jednak wystąpić dowolną liczbę razy wewnątrz bloku.

```
float Kwadrat ( float Wartość )
{
   return Wartość * Wartość ;
}
//
void Max ( int *Największy, int Kandydat1, int Kandydat2 )
{  *Największy = Kandydat1 > Kandydat2 ? Kandydat1 : Kandydat2 ;
 //  return;                  // zbędne
}
```

```
int Podział ( char *Tekst, char *Parzyste, char *Nieparzyste )
{
    int LiczbaZnaków = 0 ;
    while ( *Tekst )
    {
      if ( LiczbaZnaków & 0x01 )
          *Nieparzyste++ = *Tekst++ ;
      else
          *Parzyste++ = *Tekst++ ;
      LiczbaZnaków++ ;
    }
    *Parzyste = *Nieparzyste = 0 ;          // koniec ciągów znaków
    return LiczbaZnaków ;
}
```

Funkcja Podział rozdziela ciąg znaków wskazywany przez argument Tekst na dwa ciągi – znaki z pozycji parzystych przepisywane są do ciągu o wskaźniku Parzyste, znaki z pozycji nieparzystych do ciągu o wskaźniku Nieparzyste. Wynikiem funkcji jest łączna liczba przepisanych znaków (czyli długość ciągu znaków wskazanego przez pierwszy argument).

## 7.4. Wywołanie funkcji

*Wywołanie funkcji* powoduje przekazanie sterowania do podprogramu zapisanego jako definicja tej funkcji. Wywołanie funkcji ma postać :

*identyfikator_funkcji* ( *lista_argumentów_aktualnych* )

gdzie *lista_argumentów_aktualnych* jest ciągiem oddzielonych przecinkami wyrażeń. Wartości kolejnych wyrażeń są wiązane z kolejnymi argumentami formalnymi funkcji.

```
int Suma (int a, int b, int c)
{
  return a + b + c ;
}
//
int x = 5, y ;
y = Suma ( x, x + 9, 17 ) ;
```

Podane wywołanie funkcji Suma spowoduje przypisanie jej argumentom formalnym następujących wartości:

a ← 5 , b ← 14 , c ← 17

Wywołanie funkcji jest wyrażeniem elementarnym i może być użyte do tworzenia wyrażeń i instrukcji.

```
Kwadrat  ( 1.25 ) + 1.347                          // wyrażenie
Max ( &Lepsza,  CenaPierwsza, CenaDruga ) ;        // instrukcja
```

### 7.4.1. Przekazywanie argumentów funkcji

Analizując wywołanie funkcji, kompilator sprawdza, czy typ wartości wyrażenia jest zgodny z typem odpowiadającego mu argumentu formalnego funkcji. Zgodne są typy identyczne i takie typy, dla których można dokonać automatycznej konwersji wartości. Warunek zgodności spełniają więc wszystkie typy liczbowe i znakowe, nie spełniają go wskaźniki danych różnych typów.

```
long  DodajTysiąc ( long  Dana )
{
    return  Dana  +  1000L ;
}
DodajTysiąc ( 3 ) ;                  // poprawnie int → long
DodajTysiąc ( 1543467543L ) ;        // poprawnie, bez konwersji
DodajTysiąc ( 17.8 ) ;               // poprawnie double → long
DodajTysiąc ( 'A' ) ;                // poprawnie char → long
DodajTysiąc ( "ALA" ) ;              // błąd, konwersja niemożliwa
//
int Oblicz (float  *AdresDanejRzeczywistej ) ;
int  *AdresDanejCałkowitej ;
Oblicz ( AdresDanejCałkowitej ) ;  // błąd, nieodpowiedni wskaźnik
```

W języku C istnieją trzy sposoby przekazywania funkcjom argumentów:

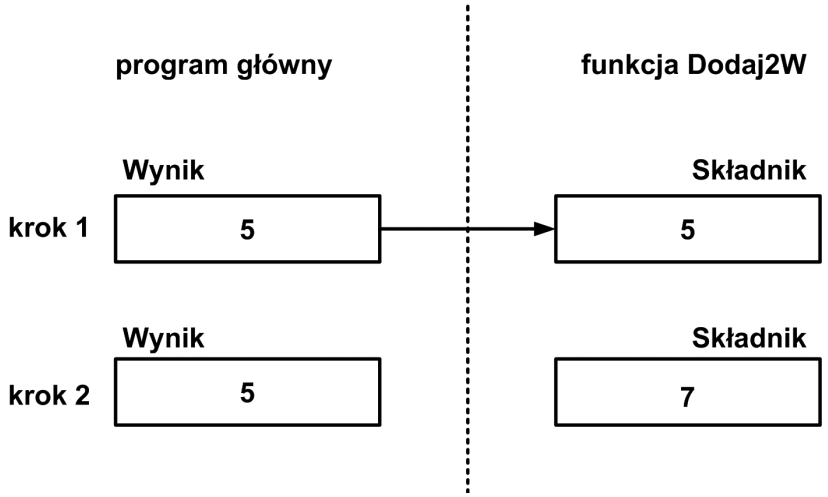
- przekazywanie wartości,
- przekazywanie wskaźnika,
- przekazywanie referencji.

Przekazywanie wartości jest stosowane przede wszystkim dla argumentów typów liczbowych. W procesie wywoływania funkcji tworzy się pomocniczą zmienną lokalną, o identyfikatorze będącym identyfikatorem parametru formalnego i wartości podanej w wywołaniu funkcji. Podprogram wywoływany dla obliczenia wartości funkcji korzysta wyłącznie z tej zmiennej pomocniczej. Po zakończeniu jego wykonywania pomocnicza zmienna lokalna jest kasowana.

```
void Dodaj2W ( int  Składnik )
{
   Składnik  =  Składnik  +  2 ;          // krok 2
}
//
```

```
int  Wynik  =  5 ;
Dodaj2W ( Wynik ) ;                    // krok 1
```

W kroku 1 (rys. 7.1.) następuje przekazanie wartości zmiennej Wynik lokalnej zmiennej Składnik, która w kroku 2 jest zwiększana o 2. Wartość zmiennej Wynik programu głównego nie ulega zmianie.




Rys. 7.1. Przekazywanie wartości

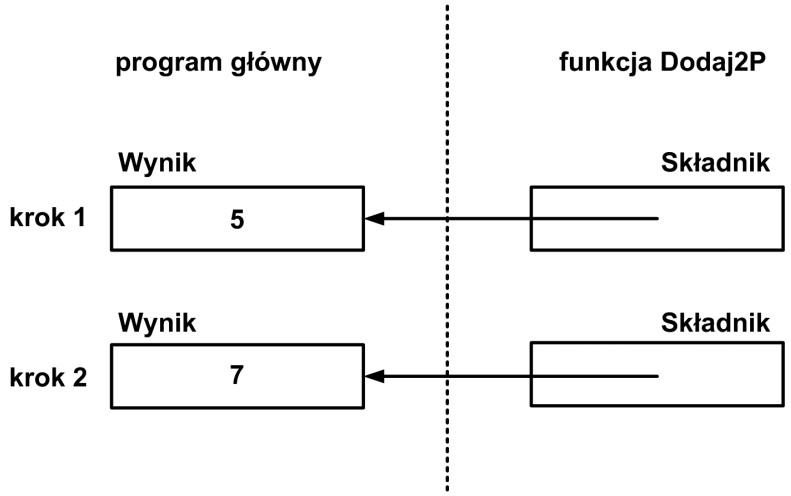
Przekazywanie wartości, jako jedyne spośród metod przekazywania funkcjom argumentów, umożliwia użycie jako argumentu aktulanego wyrażenia (a nie tylko identyfikatora zmiennej czy elementu struktury danych)

```
Dodaj2W ( Wynik * Wynik + 5 ) ;
```

Przekazywanie wskaźnika umożliwia zmianę wartości zmiennej (lub elementu struktury danych) znajdującej się w programie głównym. W tym przypadku również tworzy się pomocniczą zmienną o identyfikatorze parametru formalnego i wartości będącej wskaźnikiem danej z programu głównego.

```
void  Dodaj2P ( int  *Składnik )
{
    *Składnik  =  *Składnik  +  2 ;          // krok 2
}
//
int  Wynik  =  5 ;
Dodaj2P ( &Wynik ) ;                     // krok 1
```

Po wykonaniu funkcji Dodaj2P (rys. 7.2.) zmienna Wynik programu głównego będzie miała wartość 7 – zmiana wartości tej zmiennej jest efektem ubocznym wykonania funkcji Dodaj2P.




Rys. 7.2. Przekazywanie wskaźnika

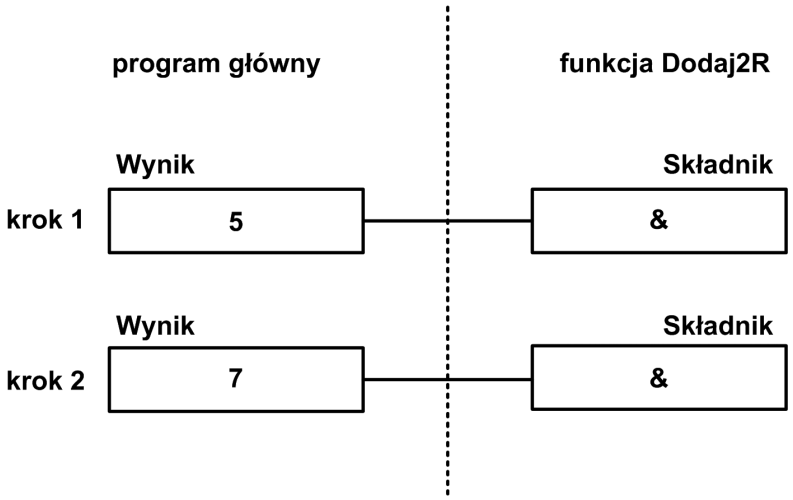
Zmianę wartości zmiennej programu głównego można również uzyskać po zdefiniowaniu funkcji, której argumentem jest referencja.

```
void Dodaj2R ( int  &Składnik )
{
    Składnik  =  Składnik  +  2 ;              // krok 2
}
//
int  Wynik  =  5 ;
Dodaj2R ( Wynik ) ;                            // krok 1
```

Tymczasowa zmienna lokalna Składnik (rys. 7.3.) staje się w tym wypadku referencją (zastępcą) zmiennej Wynik. Ponieważ wszystkie operacje wykonane na referencji zmiennej mają skutki identyczne z operacjami wykonanymi bezpośrednio na zmiennej (punkt 3.12), uzyskana zostanie zmiana wartości zmiennej Wynik programu głównego, jako efekt uboczny wykonania funkcji Dodaj2R.

Właściwości referencji powodują jednak niemożność wywołania funkcji z argumentem referencyjnym w przypadku, gdy typ argumentu formalnego nie jest taki sam, jak typ wartości przekazywanej w wywołaniu funkcji.

```
double  Duży  =  5.0 ;
Dodaj2R ( Duży ) ;                 // błąd
```




Rys. 7.3. Przekazywanie referencji

W deklaracji funkcji można przed jej argumentem formalnym, będącym wskaźnikiem lub referencją, umieścić słowo kluczowe const. Wykonanie funkcji na pewno nie spowoduje w tym przypadku zmiany wartości zmiennej, której wskaźnik lub referencja występuje w wywołaniu funkcji jako argument aktualny.

```
float  PoleKoła (const float ∗LiczbaPi,  float  ∗Promień )
{     ...     }
float  Pi  =  3.1415 ;
float  Promień = 4.57;
PoleKoła ( &Pi,  &Promień ) ;
```

Funkcja PoleKoła na pewno nie dokona zmiany wartości zmiennej Pi, może ona natomiast zmienić wartość zmiennej Promień. Za pomocą słowa kluczowego const, umieszczanego przed argumentami formalnymi, można więc ograniczać zasięg efektów ubocznych funkcji.

## 7.4.2. Tablice jako argumenty funkcji

Argumentem funkcji może być również tablica. Podczas wywoływania takiej funkcji przekazuje się wskaźnik początkowego elementu tablicy, a wszelkie operacje są wykonywane bezpośrednio na elementach tablicy. Dla tablic jednowymiarowych deklaracja argumentu formalnego nie musi zawierać określenia liczby elementów tablicy.

```
int  ZmianaTablicy ( int ∗ ) ;                    //  deklaracje
int  ZmianaTablicy ( int [  ] ) ;
int  ZmianaTablicy ( int [ 10 ] ) ;
```

```
//
int  ZmianaTablicy  ( int *Tablica )  { ... }                    // definicje
int  ZmianaTablicy  ( int  Tablica [ ] )  { ... }
int  ZmianaTablicy  ( int  Tablica [ 10 ] )  { ... }
```

Podane trzy sposoby deklarowania i trzy sposoby definiowania funkcji ZmianaTablicy, przetwarzającej tablicę liczb całkowitych, są równoważne. Liczba elementów tablicy będącej argumentem funkcji nie musi być podana, bowiem położenie elementu tablicy (np. Tablica [ 3 ] ) jest podczas wykonywania funkcji obliczane na podstawie wskaźnika (adresu) początkowego elementu tablicy, wartości indeksu oraz rozmiaru pojedynczego elementu tablicy:

```
&Tablica [ 0 ]  + 3 * 4                    //  sizeof ( int ) → 4
```

Znajomość liczby elementów tablicy byłaby konieczna, gdyby kompilator wprowadzał do przekładu sekwencje rozkazów sprawdzających, czy wartości indeksów nie są większe od rozmiarów tablic. Takie sekwencje sprawdzające nie są generowane, obowiązek ograniczania wartości indeksów spoczywa na programiście.

```
void ZapiszSto ( int  Tablica [ 100 ] )
{
     for ( int  Kolejny  =  0 ;  Kolejny  <  100 ;  ++Kolejny )
          Tablica [ Kolejny ]  =  Kolejny ;
}
//
int Alfa [ 15 ] ;
int Beta [ 100 ] ;
int Gamma [ 150 ] ;
```

Zastosowanie funkcji ZapiszSto do tablicy Alfa spowoduje zmianę wartości 340 bajtów ( (100 – 15) * 4 ) znajdujących się za obszarem pamięci zajętym przez tę tablicę. Może to spowodować zniszczenie przechowywanych tam wartości i dezorganizację programu. Tablicę Beta funkcja ZapiszSto wypełni w całości, tablica Gamma zostanie wypełniona w 2/3. Bardziej bezpieczne jest przekazywanie funkcji rozmiaru tablicy jako odrębnego argumentu.

```
void Zapisz ( int Tablica [ ],  int Rozmiar )
{
   for ( int Kolejny  =  0 ; Kolejny  <  Rozmiar ; ++Kolejny )
        Tablica [ Kolejny ]  =  Kolejny ;
}
```

W wywołaniu funkcji, której argumentem jest tablica, jako argument aktualny musi wystąpić wskaźnik wartości takiego typu, jaki podany został w deklaracji argumentu formalnego.

```
float  Średnia ( float TablicaPomiarów [ ],  int Rozmiar ) { ... }
```

```
//
float  Pomiary [ 50 ] ;
Średnia ( Pomiary,  50 ) ;
Średnia (&Pomiary [ 10 ],  40) ;
```

Gdy argumentem funkcji jest tablica wielowymiarowa, to w definicji funkcji muszą być podane liczby elementów wszystkich wymiarów tablicy, poza najstarszym (pierwszym z lewej). Dla tablicy dwuwymiarowej należy więc podać liczbę elementów w wierszu (czyli liczbę kolumn).

```
void ZapiszMacierz ( float Macierz [  ] [ 10 ],  int Wierszy )
{
   for (int Wiersz  =  0 ; Wiersz  <  Wierszy ; ++Wiersz )
      for (int  Kolumna  =  0 ;  Kolumna < 10 ; ++Kolumna )
         Macierz [ Wiersz ] [ Kolumna ]  =  Wiersz + Kolumna ;
}
```

Liczba elementów młodszych wymiarów tablicy jest niezbędna do obliczenia położenia wskazanego indeksami elementu tablicy. Dla elementu znajdującego się w trzecim wierszu i piątej kolumnie Macierz [ 3 ] [ 5 ] będzie to:

```
&Macierz [ 0 ] [ 0 ]  +  3 * 10 * 4 + 5 * 4        //  sizeof ( float ) → 4
```

gdzie 4 jest liczbą bajtów zajmowanych przez pojedynczy element tablicy. Funkcja ZapiszMacierz powinna być stosowana wyłącznie do tablic dwuwymiarowych typu float , które mają po 10 elementów w wierszu.

### 7.4.3. Wartości domyślne argumentów funkcji

W deklaracji i w definicji funkcji można podać wartości domyślne argumentów tej funkcji. W wywołaniu takiej funkcji mogą wówczas nie występować argumenty aktualne odpowiadające argumentom formalnym, dla których zdefiniowano wartości domyślne. Pominięte być mogą jedynie wartości znajdujące się na końcu listy argumentów aktualnych funkcji.

```
float SumaTrzech ( float Pierwszy, float  Drugi = 2.5, float Trzeci = 3.5 )
{
   return  Pierwszy  +  Drugi  +  Trzeci ;
}
//
float  Alfa = 2.5,  Beta = 8.9, Gamma = 19.2 ;
SumaTrzech ( Alfa, Beta, Gamma ) ;          // Alfa + Beta + Gamma
SumaTrzech (Alfa, Beta ) ;                  // Alfa + Beta + 3.5
SumaTrzech ( Alfa ) ;                       // Alfa + 2.5 + 3.5
SumaTrzech (  ) ;                           // błąd
SumaTrzech (Alfa,  ,Gamma ) ;               // błąd
```

### 7.4.4. Funkcje rekurencyjne

W bloku definiującym funkcję może wystąpić wywołanie definiowanej funkcji, dzięki czemu jest możliwe tworzenie definicji rekurencyjnych.

```
int IloczynRekurencyjny ( int Liczba )
{
   return  Liczba == 1 ? 1 : Liczba * IloczynRekurencyjny  (Liczba  – 1 ) ;
}
```

Funkcja rekurencyjna IloczynRekurencyjny dla liczby naturalnej n oblicza wartość iloczynu

$$n * ( n – 1 ) * ( n – 2 ) * ... * 1$$

Tę samą wartość można również wyznaczyć za pomocą funkcji iteracyjnej.

```
int IloczynIteracyjny  ( int Liczba )
{
   int  Iloczyn  =  1 ;
   while  ( Liczba  !=  1 )
   {
      Iloczyn  *=  Liczba ;
      – –Liczba ;
   }
   return  Iloczyn ;
}
```

Tekst definicji funkcji IloczynRekurencyjny jest krótszy, jednak czas wykonania i wielkość zajętego obszaru pamięci będą większe niż dla funkcji IloczynIteracyjny. Przy każdym kolejnym wywołaniu funkcji IloczynRekurencyjny tworzony jest obszar roboczy, do którego wpisywana jest między innymi aktualna wartość argumentu Liczba. Wszystkie utworzone dla kolejnych wywołań funkcji IloczynRekurencyjny obszary robocze są kasowane dopiero po uzyskaniu przez argument Liczba wartości równej 1.

### 7.4.5. Podprogramy wstawiane

Jeżeli przed definicją funkcji występuje słowo kluczowe inline, to przekład tej funkcji będzie przez kompilator umieszczany w programie głównym, w miejscach, w których występują jej wywołania. Dzięki takiemu rozwiązaniu można zaoszczędzić czas zużywany na wywołanie funkcji i powrót do programu głównego. Metoda ta może być więc polecana dla funkcji o krótkich definicjach lub dla funkcji wywoływanych niewielką liczbę razy.

```
inline double  Wybór ( double  Moje,  double Twoje,  double  Nasze )
{     return  Moje >  Twoje  ?   Nasze  :  Moje ;      }
```

## 7.5. Wskaźniki funkcji

Rozważmy, w jaki sposób zdefiniować funkcję Zastosuj, która w zależności od wartości swego argumentu wywoła jedną z funkcji ObliczJeden ( ), ObliczDwa ( ), itd. należących do zestawu Obliczenia. Niewątpliwie można funkcje zestawu Obliczenia ponumerować, numer funkcji do wywołania przekazać jako argument typu int, a w bloku definiującym funkcję Zastosuj umieścić instrukcję wyboru sterowaną wartością tego argumentu.

```
void Zastosuj ( int  NumerFunkcji )
{
   switch ( NumerFunkcji )
      {
      case 1 :    ObliczJeden ( ) ;
                  break;
      case 2 :    ObliczDwa ( ) ;
                  break;
     // . . . . . . . . . . . . . .
   }
}
```

Rozwiązanie takie jest jednak niekorzystne, gdyż wywołanie każdej z funkcji zestawu Obliczenia musi bezpośrednio wystąpić w funkcji Zastosuj. Funkcja ta jest wobec tego ściśle związana z zestawem Obliczenia i musi być modyfikowana każdorazowo, po zmianie liczby funkcji należących do tego zestawu. Właściwym rozwiązaniem byłoby przekazanie funkcji Zastosuj informacji opisującej funkcję, która ma zostać wywołana. W języku C identyfikator funkcji nie może być użyty jako argument innej funkcji, można natomiast jako argumentu użyć *wskaźnika funkcji*. Wskaźnik funkcji, podobnie jak wskaźnik zmiennej określonego typu, zawiera informację o lokalizacji podprogramu zdefiniowanego za pomocą funkcji (czyli jego adres) oraz informację o typie wyniku i sygnaturze funkcji. Deklaracja:

```
int  ( *Funkcja ) ( int, int ) ;
```

stwierdza więc, że wartością zmiennej o identyfikatorze Funkcja będzie wskaźnik funkcji o wyniku typu int i dwu argumentach, obydwu typu int. Wartościami zmiennej Funkcja mogą być wyłącznie wskaźniki tak zdefiniowanych funkcji.

```
int  ObliczCałkowitą ( int Pierwszy,  int Drugi )
{
      return ( Pierwszy + Drugi ) * ( Pierwszy – Drugi ) ;
}
long  ObliczDługą ( int Pierwszy,  int  Drugi )
{
      return  (long ( ( Pierwszy + Drugi ) * ( Pierwszy – Drugi ) ) ;
}
```

```
Funkcja = ObliczCałkowitą ;        // poprawnie
Funkcja = &ObliczCałkowitą ;       // poprawnie
Funkcja = ObliczDługą ;            // błąd, nieodpowiednia funkcja
```

Zmiennym, których wartościami są wskaźniki funkcji można nadawać wartości początkowe.

```
void SortowanieBąbelkowe ( float TablicaDanych [ ], int Rozmiar )
{ ... }
void SortowaniePołączeniowe ( float TablicaDanych [ ], int Rozmiar )
{ ... }
void SortowanieStertowe ( float TablicaDanych [ ], int Rozmiar )
{ ... }
//
void ( *SortowanieJeden ) ( float [ ], int ) = SortowanieBąbelkowe ;
void ( *SortowanieDwa ) ( float [ ], int ) ;
SortowanieDwa = SortowanieJeden ;
```

Wywołanie funkcji za pośrednictwem zmiennej, której wartością jest wskaźnik tej funkcji, nie wymaga stosowania operatora dostępu pośredniego.

```
float Pomiary [ 150 ] ;
//
SortowanieJeden ( Pomiary, 150 ) ;
( *SortowanieJeden ) ( Pomiary, 150 ) ;
/* równoważne wywołania funkcji SortowanieBąbelkowe */
```

### 7.5.1. Tablice wskaźników funkcji

Wskaźniki funkcji mogą również być elementami tablic. Tablicę wskaźników funkcji deklaruje się, podając za identyfikatorem tablicy liczbę jej elementów (rozmiar), np.:

```
int ( *Test [ 10 ] ) ( char* ) ;
```

Tablica Test zawiera 10 elementów ponumerowanych od 0 do 9 będących wskaźnikami funkcji o wyniku typu int i pojedynczym argumencie typu char* (wskaźnik ciągu znaków). Nadawanie wartości elementom takiej tablicy może być realizowane za pomocą operatora przypisania = lub w definicji tablicy.

```
int FunkcjaAlfa ( double Zakres )
{ ... }
int FunkcjaBeta ( double Zakres )
{ ... }
int FunkcjaGamma ( double Zakres )
{ ... }
int ( *TrzyFunkcje [ 3 ] ) ( double ) = { FunkcjaAlfa, FunkcjaBeta } ;
```

```
TrzyFunkcje [ 2 ]  =  FunkcjaGamma ;
```

Wywołanie funkcji, której wskaźnik jest wartością elementu tablicy może mieć na przykład postać:

```
TrzyFunkcje [ 1 ] ( 37.18 ) ;
```

Natomiast wykonanie wszystkich funkcji wskazywanych przez elementy tablicy Test [10] uzyskać można za pomocą pętli:

```
int  Wyniki [ 10 ] ;
char* Teksty [ 10 ] ;
/* ... */
for ( int  Licznik  =  0 ;  Licznik  <  10 ;  ++Licznik )
   Wyniki [ Licznik ] = Test [ Licznik ] ( Teksty [ Licznik ] ) ;
```

Zapis definicji zmiennych i tablic, których wartościami są wskaźniki funkcji można uprościć, korzystając z możliwości nazywania typów danych. Zapis:

```
typedef  int ( *FunkcjePierwotne ) ( float,  float ) ;
```

wprowadza typ FunkcjePierwotne, do którego należą wskaźniki funkcji o wyniku typu int i dwu argumentach typu float. Za pomocą identyfikatora FunkcjePierwotne można następnie deklarować zmienne i tablice.

```
int  Przemiana ( float  Start,  float  Stop )
{  ...  }
//
FunkcjePierwotne  Pierwsza,  Druga  =  Przemiana ;
FunkcjePierwotne ZestawFunkcjiPierwotnych [ 15 ] ;
```

### 7.5.2 Wskaźnik funkcji jako argument funkcji

Rozwiązanie problemu rozważanego na początku tego punktu wymaga użycia wskaźnika funkcji jako argumentu funkcji. Deklaracja parametrycznej funkcji sortującej, wywołującej jedną ze specyficznych funkcji sortujących, może mieć na przykład postać:

```
void  Sortuj ( float [  ] ,  int ,  void  (*)  ( float [ ], int )   =
                                        SortowaniePołączeniowe ) ;
```

Trzeci argument funkcji Sortuj jest typu: wskaźnik funkcji o wartości typu void i dwu argumentach: pierwszym jest jednowymiarowa tablica liczb zmiennopozycyjnych, drugim wartość całkowita. Argument ten ma nadaną wartość początkową, będącą wskaźnikiem funkcji SortowaniePołączeniowe. Definicja funkcji Sortuj może mieć postać:

```
void Sortuj ( float  Dane [  ], int Rozmiar,       void ( *Wykonawca ) ( float [  ], int ) )
{
   Wykonawca ( Dane,  Rozmiar ) ;
}
```

W wywołaniu funkcji Sortuj jako trzeci argument wystąpić może identyfikator lub wskaźnik jednej ze specyficznych funkcji sortujących (gdy trzeci argument nie jest podany w wywołaniu stosuje się wartość domyślną wskazującą funkcję SortowaniePołączeniowe).

```
float  DanePoczątkowe [ 100 ] ;
//
void  ( *Sortowanie ) ( float [ ], int )  =  SortowanieStertowe ;
Sortuj ( DanePoczątkowe,  100,  SortowanieStertowe ) ;
Sortuj ( DanePoczątkowe,  100,  Sortowanie ) ;
Sortuj ( DanePoczątkowe,  100,  *Sortowanie ) ;
/* w każdym z trzech wywołań funkcji Sortuj zastosowana zostanie funkcja
   SortowanieStertowe */
```

### 7.5.3. Wskaźnik funkcji jako wynik funkcji

Wynikiem funkcji może być również wskaźnik funkcji. Dla zadeklarowania takiej funkcji wygodnie jest zdefiniować najpierw typ wskaźnikowy, np.:

```
typedef  char* ( *PrzetwarzanieŁańcuchów ) ( char*,  char* ) ;
```

Identyfikator PrzetwarzanieŁańcuchów oznacza typ, do którego należą wskaźniki funkcji o wyniku będącym wskaźnikiem ciągu znaków ( char* ) i dwu argumentach będących również wskaźnikami ciągów znaków. Funkcja o deklaracji:

```
PrzetwarzanieŁańcuchów  DobórFuncji ( char*,  int ) ;
```

posiada więc jako wynik wskaźnik funkcji typu PrzetwarzanieŁańcuchów. Przykładem może być funkcja Szukaj, przeglądająca tablicę struktur w celu wybrania funkcji, która ma cechę o zadanej wartości.

```
struct Funkcje
{
    int  Cecha ;
    PrzetwarzanieŁańcuchów  Funkcja ;
} ;
PrzetwarzanieŁańcuchów  Szukaj ( Funkcje TablicaStruktur [ ],  int  Rozmiar,  int  Wzorzec )
{
     for  ( int  Kolejny  =  0 ;  Kolejny  <  Rozmiar ;  ++Kolejny )
          if  ( TablicaStruktur [ Kolejny ] . Cecha  ==  Wzorzec )
              return  TablicaStruktur [ Kolejny ] . Funkcja ;
     return  NULL ;
}
//
Funkcje  ZestawFunkcji [ 50 ] ;
//
PrzetwarzanieŁańcuchów  Wybrana  =  Szukaj ( ZestawFunkcji,  50, 15 ) ;
```

## 7.6. Przeciążanie identyfikatorów funkcji

Rozważmy sytuację, w której taki sam algorytm ma być zastosowany do danych różnych typów. Korzystając z dotychczas przedstawionych konstrukcji języka C, należałoby w tym przypadku zdefiniować kilka funkcji o różnych identyfikatorach i za pomocą instrukcji warunkowej lub instrukcji wyboru wywołać funkcję o identyfikatorze odpowiednim do typu aktualnie przetwarzanych danych.

```
float  ŚredniaCałkowitych ( int TablicaLiczb [  ],  int  Rozmiar )
{
    float  Suma  =  0.0F;
    for ( int  Kolejny  =  0 ;  Kolejny  <  Rozmiar  ;  ++Kolejny )
      Suma  +=  (float) TablicaLiczb [ Kolejny ] ;
    return  Suma / (float) Rozmiar ;
}
//
double  ŚredniaRzeczywistych  ( double TablicaLiczb [  ],
                                        int  Rozmiar )
{
    double  Suma  =  0.0 ;
    for ( int  Kolejny = 0 ; Kolejny  <  Rozmiar ;  ++Kolejny )
        Suma  +=  TablicaLiczb [ Kolejny ] ;
   return  Suma / (double) Rozmiar ;
}
//
void ŚredniaArytmetyczna ( void * Tablica,  int Typ,
                                int  Rozmiar,  void  *wartość)
{
  switch ( Typ )
  {
    case 2   :  *( int* ) wartość  =  ŚredniaCałkowitych ( ( int* ) Tablica,  Rozmiar ) ;
                                        break;
    case 10 :  *( double* ) wartość  =   ŚredniaRzeczywistych  ( ( double* ) Tablica,  Rozmiar);
                                        break;
  }
}
```

Możliwość *przeciążania identyfikatora funkcji* pozwala na zdefiniowanie kilku funkcji o takim samym identyfikatorze, lecz o różnych sygnaturach (czyli różnych listach typów argumentów). Po napotkaniu wywołania funkcji o przeciążonym identyfikatorze kompilator dobiera taką jej wersję, jaka najlepiej odpowiada typom argumentów aktualnych.

```
float  Średnia ( int TablicaLiczb [  ],  int  Rozmiar )
{
    float  Suma  =  0.0F ;
    for ( int  Kolejny  =  0 ;  Kolejny  <  Rozmiar  ;  ++Kolejny )
      Suma  +=  (float) TablicaLiczb [ Kolejny ] ;
    return  Suma / (float) Rozmiar ;
}
double  Średnia  ( double TablicaLiczb [  ],  int  Rozmiar )
{
    double  Suma  =  0.0 ;
    for ( int  Kolejn = 0 ; Kolejny  <  Rozmiar ; ++Kolejny )
        Suma  +=  TablicaLiczb [ Kolejny ] ;
   return  Suma / (double) Rozmiar ;
}
//
Średnia ( TablicaCałkowitych,  Rozmiar ) ;
Średnia ( TablicaRzeczywistych,  Rozmiar ) ;
```

Funkcje o takim samym przeciążonym identyfikatorze muszą mieć różne sygnatury. Jeżeli sygnatury są równe, to:

- przy równych typach wyniku funkcji zachodzi przesłanianie funkcji wcześniej zdefiniowanej (np. w innym pliku) przez funkcję zdefiniowaną później,
- przy różnych typach wyniku funkcji kompilator sygnalizuje błąd.

```
 /* plik funkcji bibliotecznych */
int  strlen ( char *Łańcuch )
{  ...  }
/* plik programu użytkowego */
int  strlen ( char *Znaki )
{
   int  Długość  =  0 ;
   while ( *Znaki++ && *Znaki != ' ' )
          Długość++ ;
   return  Długość ;
}
/* definicja zawarta w pliku programu użytkowego przesłania
  definicję z pliku bibliotecznego */
//
void czytaj ( char *  Alfa )
{ ... }
int czytaj ( char * Beta )                  // błąd
{  ...  }
```

Należy przy tym pamiętać, że nadanie typowi danych innej nazwy nie wprowadza nowego typu, a jedynie wiąże z dotychczas istniejącym typem dodatkowy identyfikator.

```
typedef  int  num ;
int  maximum ( int,  int )  { ... };
char maximum ( num,  num ) { ... } ;   // błąd
```

Przykładem poprawnie zdefiniowanych funkcji o przeciążonym identyfikatorze może być następująca grupa funkcji wyprowadzających.

```
void  drukuj ( char Znak )          // wersja C
{  ...  }
void  drukuj ( int Liczba )         // wersja I
{  ...  }
void drukuj ( float Liczba )        // wersja F
{  ...  }
void drukuj ( char *Ciąg )          // wersja S
{  ...  }
```

Po napotkaniu wywołania funkcji o przeciążonym identyfikatorze, kompilator analizuje listę typów argumentów aktualnych i dobiera odpowiednią wersję funkcji. Podczas doboru wersji dla każdego argumentu mogą wystąpić następujące przypadki.

| | |
|---|---|
| zgodność | argument aktualny jest takiego samego typu co argument formalny lub wartość argumentu aktualnego może zostać przekształcona za pomocą standardowej konwersji na wartość takiego typu, jaki ma argument formalny. |
| niezgodność | za pomocą standardowej konwersji wartość argumentu aktualnego nie może zostać przekształcona na wartość typu określonego dla argumentu formalnego. |
| niejednoznaczność | argument aktualny jest zgodny z argumentem formalnym co najmniej dwu funkcji. |

Po wykryciu niezgodności lub niejednoznaczności kompilator sygnalizuje błąd. W przypadku niejednoznaczności postępowanie takie wynika z równoprawnego traktowania wszystkich konwersji, bez wyróżniania konwersji lepszych i gorszych.

```
int  Wynik = 15 ;
drukuj ( Wynik ) ;          // zgodność: wersja I
drukuj ( 'A' ) ;            // zgodność: wersja C
drukuj ( "Wynik" ) ;        // zgodność: wersja S
```

```
float  Liczba, *Wskazanie  =  &Liczba ;
drukuj ( Liczba ) ;            // zgodność: wersja F
drukuj ( Wskazanie ) ;         // błąd: niezgodność
double  Cena ;
drukuj ( Cena ) ;              // błąd: niejednoznaczność – możliwe wersje C, I, F
```

Podczas doboru wersji funkcji o przeciążonym identyfikatorze kompilator stosuje zasadę minimalizacji liczby konwersji, którym należy poddać wartości argumentów aktualnych. Zasada ta jest szczególnie widoczna w przypadku funkcji o więcej niż jednym argumencie. I tak dla funkcji o przeciążonym identyfikatorze Frekwencja

```
void   Frekwencja ( int  Liczba,  float Współczynnik )      // wersja 1
{  ...  }
void  Frekwencja ( char Znak,  float Współczynnik )         // wersja 2
{  ...  }
```

do realizacji wywołania

```
Frekwencja ( 'a', 15 ) ;
```

zostanie wybrana wersja 2 funkcji Frekwencja, która wymaga tylko jednej konwersji ( int → float ). Wersja 1 funkcji Frekwencja wymagałaby natomiast dwu konwersji ( char → int , int → float ).

Argumentom funkcji o przeciążonym identyfikatorze można również nadawać wartości domyślne.

```
void  Oblicz ( int  Dana )                          // wersja 1
{  ...  }
void  Oblicz ( long Liczba,  int Rozmiar  =  0 )    // wersja 2
{  ...  }
//
Oblicz ( 2L );          // wersja 2: Oblicz ( 2L, 0 ) ;
Oblicz ( 5, 38 ) ;      // wersja 2: Oblicz ( 5, 38 ) ;
Oblicz ( 137 ) ;        // wersja 1: Oblicz ( 137 ) ;
Oblicz ( 3.14 ) ;       // błąd: niejednoznaczność
```

Wszystkie funkcje o takim samym przeciążonym identyfikatorze powinny być definiowane w tym samym zakresie widoczności, czyli w tym samym pliku.

### 7.6.1. Wzorce funkcji

Definicje kilku funkcji o przeciążonym identyfikatorze zawierają niekiedy dokładnie taki sam blok określający sposób obliczania wartości funkcji.

```
int  minimum ( int Pierwsza,  int  Druga)                    // wersja I
{
    return  Pierwsza  <  Druga  ?  Pierwsza  :  Druga ;
}
```

```
long minimum ( long Pierwsza,  long Druga )                    // wersja L
{
    return  Pierwsza  <  Druga  ?  Pierwsza  :  Druga ;
}
float minimum ( float Pierwsza,  float  Druga )                // wersja F
{
    return  Pierwsza  <  Druga  ?  Pierwsza  :  Druga ;
}
```

Kompilator języka C umożliwia zapisanie takiego zestawu funkcji za pomocą wzorca funkcji.

```
template  <class T>  T  minimum ( T  Pierwsza, T Druga )
{
    return  Pierwsza  <  Druga  ?  Pierwsza  :  Druga ;
}
```

Zapis ten sygnalizuje kompilatorowi, że stosownie do listy typów argumentów aktualnych ma wytworzyć odpowiednią wersję funkcji minimum, a więc dla typu int wersję I, dla typu float wersję F itp. Podczas wywołania funkcji o przeciążonym identyfikatorze, zdefiniowanej za pomocą wzorca, dokonywana jest konwersja wartości argumentów aktualnych.

```
minimum ( 1, 2 ) ;        // poprawnie: wersja I
minimum ( 3, 1.5 ) ;      // poprawnie: wersja I z konwersją double –> int
```

## 7.7. Funkcje biblioteczne

Systemy programowania w języku C udostępniają najczęściej kilkaset *funkcji bibliotecznych* ułatwiających pisanie programów i wspomagających zarządzanie zasobami komputera. W zestawie funkcji bibliotecznych występują zazwyczaj następujące grupy funkcji:

| | |
|---|---|
| czas | odczyt daty, czasu astronomicznego i czasu względnego oraz przetwarzanie struktur danych zawierających informację o czasie, |
| ciągi znaków | przetwarzanie ciągów znaków ASCII i manipulowanie obszarami pamięci operacyjnej, |
| diagnostyka | sprawdzanie wykonalności pewnych fragmentów programu i sygnalizowanie błędów, |
| grafika | funkcje grafiki punktowej umożliwiające wyświetlanie i przetwarzanie obrazów na ekranie monitora, |

| | |
|---|---|
| katalogi | operowanie katalogami plików dyskowych i ścieżkami dostępu, |
| klasyfikacja | określanie przynależności znaków kodu ASCII do odpowiednich grup (litery, cyfry, znaki sterujące itp.), |
| konwersja | przekształcanie ciągów znaków kodu ASCII na wartości binarne i odwrotnie, zamiana liter małych na duże i odwrotnie, |
| matematyka | realizacja wielu funkcji matematycznych pomocnych w obliczeniach numerycznych, |
| tekst | sterowanie ekranem monitora w trybie tekstowym niekiedy z możliwością otwierania i przetwarzania okienek, |
| pamięć | wybór trybu adresowania pamięci, dynamiczny przydział pamięci, |
| procesy | tworzenie współbieżnych procesów obliczeniowych i zarządzanie ich wykonywaniem, |
| standard | podstawowe funkcje dokonujące konwersji znakowo-binarnej i binarno-znakowej oraz sterujące wykonywaniem programu, |
| system | współpraca z systemem operacyjnym w zakresie bezpośredniego sterowania urządzeniami zewnętrznymi, obsługi przerwań itp., |
| wejście – wyjście | wprowadzanie i wyprowadzanie danych w sposób tradycyjny i za pomocą strumieni wejściowo-wyjściowych. |

Szczegółowe opisy funkcji bibliotecznych można znaleźć na przykład w dokumentacji poszczególnych systemów programowania w języku C. Podstawowe funkcje wprowadzania i wyprowadzania danych przedstawione są w rozdziale 9.

# 8. Struktura programu

## 8.1. Części programu

Program zapisany w języku C składa się zazwyczaj z następujących części:

- dyrektyw preprocesora,
- deklaracji i definicji globalnych danych i struktur danych,
- deklaracji i definicji funkcji,
- definicji funkcji main.

Z wymienionych części w programie wystąpić musi definicja funkcji main. Wykonywanie programu rozpoczyna się bowiem zawsze od początku bloku występującego w definicji tej funkcji. Definicje innych funkcji poprzedzają zazwyczaj definicję funkcji main. Definicje funkcji mogą również być umieszczane za definicją funkcji main, o ile przed tą definicją znajdą się ich deklaracje (zapowiedzi).

## 8.2. Preprocesor

Preprocesor to program będący częścią kompilatora, który dokonuje wstępnego, tekstowego przetworzenia programu źródłowego zapisanego w języku C. Przetwarzanie to sterowane jest dyrektywami, z których każda rozpoczyna się od znaku #. Działanie niektórych dyrektyw preprocesora można zastąpić odpowiednimi konstrukcjami języka C (co nie zawsze było możliwe we wcześniejszych wersjach języka C). Omówione zostaną dyrektywy najczęściej wykorzystywane.

Dyrektywy preprocesora występują zazwyczaj na początku pliku zawierającego program źródłowy – można jednak umieszczać je również w innych miejscach programu. Każda linia programu rozpoczynająca się od znaku # będzie uważana za dyrektywę preprocesora (z wyjątkiem przypadków, w których znak # występuje na początku linii i równocześnie wewnątrz łańcucha, w stałej znakowej lub w komentarzu).

### 8.2.1. Dołączanie plików bibliotecznych

Za pomocą dyrektywy #include do programu znajdującego się we właśnie przetwarzanym pliku źródłowym dołącza się plik określony przez argument tej dyrektywy. Nazwa dołączanego pliku (lub ścieżka dostępu) ujęta być powinna w nawiasy

< >, gdy poszukiwanie wskazanego pliku ma dotyczyć standardowego katalogu INCLUDE lub w cudzysłowy " ", gdy poszukiwanie ma się rozpocząć od katalogu bieżącego.

```
#include <stdio.h>
#include "funkcje.cpp"
```

Pojedyncza dyrektywa #include umożliwia dołączanie jednego pliku; w dołączanych plikach mogą również znajdować się dyrektywy #include.

```
/*  Plik definicji funkcji  DodajOdejmij5.cpp   */
int  Dodaj5 ( int Dana )      // definicja funkcji Dodaj5
{
   return Dana  +  5 ;
}
int  Odejmij5 ( int  Dana )     // definicja funkcji Odejmij5
{
   return Dana  –  5 ;
}
//
/*  Koniec pliku DodajOdejmij5.cpp  */
//
/*  Plik programu głównego Arytmetyka.cpp  */
//
#include  "DodajOdejmij5.cpp"
//
void  main  ( void )
{
   ...              // wywołania funkcji Dodaj5 i Odejmij5
}
//
/*  Koniec pliku Arytmetyka.cpp  */
```

## 8.2.2. Zastępowanie tekstów

Dyrektywa #define powoduje zastąpienie w dalszej części programu identyfikatora będącego pierwszym argumentem tej dyrektywy przez tekst będący jej drugim argumentem (o ile ciąg znaków tworzących identyfikator nie jest częścią łańcucha, stałej znakowej lub komentarza). A więc dyrektywa

```
#define   ROZMIAR   150
```

spowoduje zastąpienie wszystkich wystąpień identyfikatora ROZMIAR (znajdujących się za tą dyrektywą) przez tekst 150. W pewnym zakresie dyrektywa ta umożliwia więc definiowanie stałych. Tekst zastępujący może składać się z wielu słów.

```
#define   moje  Cena * ( Marża  –  Procent   /  100 )
```

Wprowadzoną definicję wiążącą identyfikator i tekst zastępujący można odwołać za pomocą dyrektywy #undef, aby następnie móc z tym samym identyfikatorem związać inny tekst.

```
#define  Epsilon  3.5E-8
// ........
#undef   Epsilon
#define  Epsilon  1.5E-8
```

Tekst zastępujący może również zawierać dyrektywy #define, które są dalej interpretowane przez preprocesor. Zagnieżdżenie to nie może jednak prowadzić do rekurencji – tekst zastępujący pewien identyfikator nie może zawierać bezpośrednio lub pośrednio zastępowanego identyfikatora.

### 8.2.3. Kompilacja warunkowa

Za pomocą dyrektyw #if, #elseif, #else, #endif można sterować procesem kompilacji wskazując, które z fragmentów plików źródłowych będą przetwarzane. Dyrektywa kompilacji warunkowej ma następującą postać:

```
#if     wyrażenie_stałe_1
            tekst_1
#elseif  wyrażenie_stałe_2
            tekst_2
 // ....................................
#else
            tekst_n
#endif
```

Składniki rozpoczynające się od #elseif oraz #else są nieobowiązkowe. Jeżeli wartość *wyrażenia_stałego* (która musi być możliwa do obliczenia podczas kompilacji) jest różna od 0, to jest interpretowany związany z nią *tekst*, rozpoczynający się od nowej linii i będący ciągiem dowolnych elementów programu. W wyrażeniach stałych często jest używany operator #defined, który zastosowany do identyfikatora ma wartość:

1 gdy identyfikator był już zdefiniowany (za pomocą zinterpretowanej uprzednio dyrektywy #define),

0 gdy identyfikator jest nie zdefiniowany (ponieważ nie wystąpił w dyrektywie #define lub ostatnio wykonano dla niego dyrektywę #undef).

Skróconą formą zapisu dyrektywy

```
#if defined ( identyfikator )
```

jest

```
#ifdef   identyfikator
```

a dyrektywy

```
#if !defined ( identyfikator )
```

jest

```
#ifndef identyfikator
```

## 8.3. Funkcja main

Każdy program napisany w języku C musi zawierać definicję funkcji main. Wykonywanie programu rozpoczyna się od początku bloku definiującego tę funkcję, a zakończenie wykonywania następuje po napotkaniu instrukcji return lub po osiągnięciu końca tego bloku. Funkcja main najczęściej posiada wynik typu int (interpretowany niekiedy jako kod sposobu zakończenia programu) i zazwyczaj jest bezargumentowa.

```
int main ( void ) { ... }
```

albo

```
int main ( ) { ... }
```

Niektóre kompilatory dopuszczaja również bezwynikową postac funkcji main:

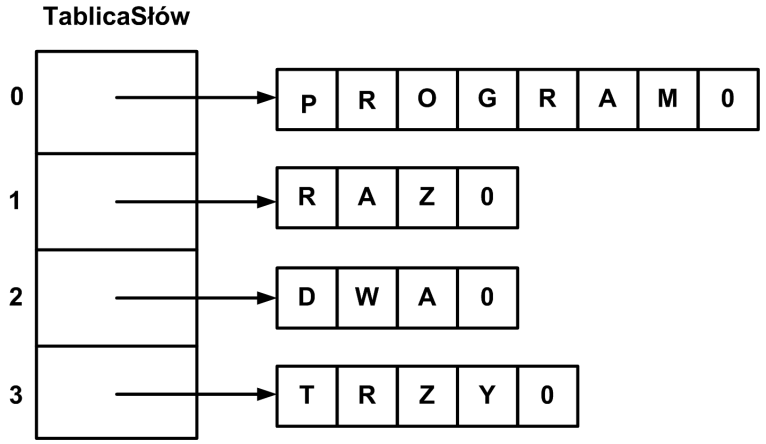
```
void main ( void ) { ... }
```

albo

```
void main ( ) { ... }
```

Aby pobrać parametry podane podczas wywołania programu, należy zdefiniować funkcję main o dwu argumentach.

```
int main ( int LiczbaSłów, char* TablicaSłów [ ] )
{ ... }
```




Rys. 8.1. Tablica parametrów wywołania programu

Po uruchomieniu programu system operacyjny przypisze parametrowi formalnemu LiczbaSłów wartość będącą liczbą słów występujących w wywołaniu programu

(łącznie z nazwą programu). Tablica wskazana przez parametr TablicaSłów zawiera natomiast wskaźniki ciągów znaków reprezentujących te słowa. Jeżeli przekład programu zapisanego w języku C znajduje się w pliku PROGRAM.EXE, to po wywołaniu PROGRAM RAZ DWA TRZY argument LiczbaSłów będzie miał wartość 4, a tablica słów będzie miała postać pokazaną na rysunku 8.1.

## 8.4. Deklaracje identyfikatorów

### 8.4.1. Zakres deklaracji

*Zakres deklaracji* identyfikatora charakteryzuje sposób jego zadeklarowania i ogólnie określa części programu, w których identyfikator ten może być użyty (jest zadeklarowany). Wyróżniamy:

zakres globalny : obejmujący cały program,

zakres lokalny : dotyczący definicji pojedynczej funkcji.

Zakres globalny mają wszystkie identyfikatory funkcji oraz identyfikatory danych i struktur danych zadeklarowane globalnie, czyli poza definicjami funkcji (najczęściej na początku programu). Zakres lokalny mają identyfikatory zadeklarowane w blokach definiujących funkcje.

```
int  Pierwszy,  Drugi,  Trzeci ;
float  Duży,  Wielki ;
int  Adaptacja( int   Dana,  int  Wzorzec )
{
    char  Znak,  Kod ;
    float  Suma ;
    // .....
}
int  Obliczenie ( float  Składnik,  float Czynnik , char Sterowanie )
{
    int  Przygotowanie [ 15 ] ;
    long  Macierz [ 15 ] [ 15 ] ;
    float  Alfa,  Beta,  Gamma ;
   // ..........................................
}
void  main ( void )
{
    int  Kolejny,  Następny, Końcowy, Ostatni ;
    float  Epsilon,  Tau,  Delta ;
    long  DaneWstępne [ 15 ] [ 15 ],  DaneKońcowe [ 15 ] [ 15 ] ;
   // ..................................................................................
}
```

W programie tym poszczególne identyfikatory mają następujące zakresy.

| | |
|---|---|
| Zakres globalny | Pierwszy, Drugi, Trzeci, Duży, Wielki, Adaptacja, Obliczenie, |
| Zakres lokalny w funkcji Adaptacja | Dana, Wzorzec, Znak, Kod, Suma, |
| Zakres lokalny w funkcji Obliczenie | Składnik, Czynnik, Sterowanie, Przygotownie, Macierz, Alfa, Beta, Gamma |
| Zakres lokalny w funkcji main | Kolejny, Następny, Końcowy, Ostatni, Epsilon, Tau, Delta, DaneWstępne, DaneKońcowe |

## 8.4.2. Zasięg deklaracji

*Zasięg deklaracji* identyfikatora obejmuje te fragmenty programu, w których obowiązuje ta deklaracja. Identyfikatory o zakresie globalnym mogą być użyte w dowolnym miejscu programu – ich zasięgiem jest więc cały program. W przypadku identyfikatorów o zakresie lokalnym obowiązuje następująca zasada: *zasięg deklaracji identyfikatora rozciąga się od miejsca wystąpienia deklaracji do końca bloku, w którym deklaracja ta wystąpiła*.

```
void main ( void )
{
  int Alfa ;
  // .............
  {
    int Beta ;
    // .................
      while ( Alfa-- )
      {
        int Delta ;
        // .................
      }
      int Jota ;
      // ................
  }
}
```

zasięg Delta

zasięg Jota

zasięg Beta

zasięg Alfa

### 8.4.3. Przesłanianie identyfikatorów

W zasięgu swego działania deklaracja identyfikatora o zakresie lokalnym przesłania deklarację tego samego identyfikatora o zakresie globalnym. W następującym przykładzie przyjęto, że w programie nie występują inne definicje lub instrukcje zmieniające wartość zmiennej Licznik.

```
int Licznik = 5 ;
void Przelicz ( )
{                                  ┐
  //.................               ├ Licznik == 5
  //.................               ┘
  int Licznik = 1 ;                ┐
  //.................               ├ Licznik == 1
}                                  ┘
void Oblicz ( )
{                                  ┐
  //.................               ├ Licznik == 5
  //.................               ┘
}
void main ( void )
{                                  ┐
  int Licznik = 0 ;                │
  //.................               │
  Przelicz ( ) ;                   │
  //.................               ├ Licznik == 0
  Oblicz ( ) ;                     │
  //.................               │
}                                  ┘
```

Wielokrotne deklarowanie tego samego identyfikatora w tym samym zakresie (globalnym albo lokalnym) jest przez kompilator sygnalizowane jako błędna redefinicja identyfikatora.

```
float  Epsilon  =  0.001 ;
//.............
double  Epsilon  =  0.05;          // błąd - redefinicja w zakresie globalnym
//.............
void  main ( void )
{
   long  Średnica ;
   //.............
   int  Średnica ;                 // błąd - redefinicja w zakresie lokalnym
   //.............
}
```

Stosowanie takich samych identyfikatorów o zakresie globalnym i lokalnym zmniejsza czytelność programu i wymaga szczegółowej analizy zasięgów działania poszczególnych deklaracji. W mniejszych programach, pisanych przez jednego programistę, nie należy więc wykorzystywać wielokrotnie tego samego identyfikatora – można przecież utworzyć wiele różnych identyfikatorów. W programach pisanych przez wielu programistów trudno niekiedy uniknąć przesłaniania identyfikatorów. Typowym przykładem są identyfikatory globalne występujące w bibliotekach funkcji, których przesłanianie zachodzi często bez wiedzy programisty.

### 8.4.4. Dostęp do obiektów globalnych

Za pomocą jednoargumentowego operatora globalizacji :: (dwa dwukropki) można uzyskać dostęp do danej globalnej o identyfikatorze przesłoniętym przez identyfikator o zakresie lokalnym. Przykładem może być funkcja wyszukująca element tablicy o wartości maksymalnej lecz nie przekraczającej wartości zadanej stałej globalnej.

```
const  int  Maksimum  =  157 ;                                  // zakres globalny
int  ZnajdźMaksimum  (int Tablica [  ],  int  Rozmiar )
{
  int Maksimum  =          INT_MIN;                             //  limits.h
  for ( int  Kolejny  =  0 ; Kolejny  <  Rozmiar ;  ++Kolejny )
    if (  Tablica [ Kolejny ]  <  ::Maksimum                    // zakres globalny
        &&
        Tablica [ Kolejny ]   >   Maksimum  )                   // zakres lokalny
      Maksimum  =  Tablica [ Kolejny ] ;                        // zakres lokalny
return Maksimum ;                                               // zakres lokalny
}
```

## 8.5. Lokalizacja zmiennych

Zmienne zadeklarowane w programie należą do jednej z następujących kategorii:

- zmienne statyczne,
- zmienne dynamiczne,
- zmienne automatyczne,
- zmienne rejestrowe.

*Zmiennymi statycznymi* są wszystkie zmienne o zakresie globalnym oraz te zmienne lokalne, które zostały poprzedzone słowem kluczowym static. Każdej zmiennej statycznej jest przydzielany odpowiedni co do długości obszar pamięci do przechowywania wartości zmiennej. Przydział ten ma charakter statyczny, czyli przydzielony obszar nie zostanie użyty do innych celów przez cały czas działania programu. Kompilator zawsze nadaje zmiennym statycznym wartość początkową – jest to wartość podana w definicji zmiennej (gdy w programie znajduje się jedynie deklaracja zmiennej, to jest jej nadawana wartość początkowa równa 0 ). Obszary pamięci przydzielone zmiennym statycznym i statycznym strukturom danych tworzą łącznie obszar danych statycznych programu. Zmienna statyczna może mieć zakres lokalny, czyli może być zadeklarowana w bloku definiującym funkcję. W takim przypadku wartość nadana zmiennej podczas kolejnego wykonania funkcji jest dostępna przy następnym wykonaniu tej funkcji. Gdy w definicji funkcji znajduje się definicja zmiennej statycznej określająca jej wartość początkową, to przypisania zmiennej tej wartości początkowej dokonuje się tylko raz, podczas pierwszego wykonania funkcji.

```
int  Licznik ( void )
{
     static  int  Licznik  =  1 ;
     return  Licznik++ ;
}
int  Numer ;
Numer  =  Licznik ( ) ;          // Numer ← 1
// ...................
// ...................
Numer  =  Licznik ( ) ;          // Numer ← 3
Numer  =  Licznik ( ) ;          // Numer ← 2
// ...................
```

Przydział pamięci do przechowywania wartości *zmiennych dynamicznych* dokonywany jest podczas wykonywania programu, w momencie napotkania deklaracji lub definicji zmiennej. Zmienne dynamiczne są więc zawsze zmiennymi o zakresie lokalnym, ograniczonym do bloku, w którym zostały zadeklarowane. Po zakończeniu wykonywania tego bloku pamięć przydzielona zmiennym dynamicznym jest

zwalniana. Zmiennym dynamicznym i dynamicznym strukturom danych przydziela się obszary pamięci na stosie programu. Deklaracja lub definicja zmiennej dynamicznej może być poprzedzona słowem kluczowym auto lub słowem register. W tym ostatnim przypadku kompilator będzie usiłował wykorzystać do przechowywania wartości zmiennej rejestry procesora. Użycie rejestrów do przechowywania wartości zmiennych powoduje przyspieszenie wykonywania programu. Poprzedzenie deklaracji lub definicji zmiennej słowem auto oddaje kompilatorowi decyzję co do lokalizacji wartości zmiennej. Słowo to jest rzadko używane, ponieważ zmienna, której deklaracja lub definicja nie jest poprzedzona żadnym z wymienionych słów kluczowych, jest zaliczana do kategorii zmiennych automatycznych.

```
register int Kolejny , Rozmiar ;
for ( Kolejny = 0 ; Kolejny < Rozmiar ; ++Kolejny )
      Tablica [ Kolejny ] = Kolejny ;
struct Lista
{
    int Cecha ;
    Lista *Następny ;
} ;
int Sprawdź (register Lista *Element, int Wzór )
{
    while ( Element != NULL )
    {
       if ( Element –> Cecha == Wzór )
          return 1;
       Element = Element –> Następny ;
    }
    return 0 ;
}
```

## 8.6. Dynamiczny przydział pamięci

Za pomocą operatora new dokonać można dynamicznego przydziału obszaru pamięci do przechowywania wartości danej lub struktury danych. Operator ten można zastosować do identyfikatora typu liczbowego, deklaracji tablicy lub identyfikatora typu struktury. Wynikiem jego działania jest wskaźnik danej lub struktury danych, utworzonej w obszarze wolnej pamięci (na stercie globalnej).

```
int *Wartość ;
Wartość = new int ;
double* Tablica = new double [ 15000 ] ;
//
```

```
struct Pies
{
    char *Smycz ;
    char *Obroża ;
    char *PiesWłaściwy ;
} ;
Pies *Morus ;
Morus = new Pies ;
```

Jeżeli obszar wolnej pamięci przeznaczony na tworzenie danych dynamicznych jest już całkowicie zajęty, to wynikiem operatora new będzie NULL.

Zwolnienie obszaru pamięci zajmowanego przez utworzoną dynamicznie daną lub strukturę danych następuje po zastosowaniu operatora delete do wskaźnika tej danej lub struktury danych.

```
delete Morus ;
delete [ ] Tablica ;
```

Dana utworzona dynamicznie za pomocą operatora new istnieje tak długo, dopóki do jej wskaźnika nie zostanie zastosowany operator delete. Możliwe jest więc dynamiczne tworzenie danych podczas wykonywania funkcji i przekazywanie wskaźników tych danych innym funkcjom.

```
const int Rozmiar = 15 ;
struct Pleks
{
    char *Etykieta ;
    float *Tablica ;
} ;
float* NowaTablica ( void )
{
    return new float [ Rozmiar ] ;
}
Pleks* NowyPleks ( char* Nazwa, float Liczba )
{
    Pleks *Nowy = new Pleks ;
    Nowy –> Etykieta = Nazwa ;
    Nowy –> Tablica = NowaTablica ( ) ;
    for ( int Kolejny = 0 ; Kolejny < Rozmiar ; ++Kolejny )
        Nowy –> Tablica [ Kolejny ] = Liczba ;
  return Nowy ;
}
```

# 9. Wprowadzanie i wyprowadzanie danych

## 9.1. Klawiatura

Po naciśnięciu jednego z klawiszy znajdujących się na klawiaturze jego numer techniczny (Dodatek B) jest przesyłany do płyty głównej komputera. Następnie procesor otrzymuje sygnał przerwania zewnętrznego, na który reaguje odpowiednia procedura systemu operacyjnego. Procedura ta dokonuje interpretacji nadesłanego numeru technicznego klawisza i opis klawisza wprowadza do obszaru pamięci zwanego buforem klawiatury. Program napisany w języku C może odczytywać zawartość bufora klawiatury za pomocą funkcji bibliotecznych. Wśród klawiszy znajdujących się na klawiaturze można wyróżnić:

| | |
|---|---|
| klawisze znakowe | oznaczone znakami należącymi do kodu ASCII, |
| klawisze sterujące | F1, F2, ... F12, Del, Home, End, PageUp, PageDown oraz klawisze oznaczone strzałkami, |
| klawisze modyfikujące | Shift, Ctrl, Alt, CapsLock, NumLock, ScrollLock, Insert . |

Klawisze modyfikujące zmieniają sposób interpretacji klawiszy znakowych i sterujących.

Różne systemy programowania w języku C posiadają różne zestawy funkcji bibliotecznych odczytujących zawartość bufora klawiatury. W punkcie tym przedstawionych zostanie kilka funkcji najczęściej spotykanych (należących do standardu języka C).

Podczas wprowadzania ciągu znaków z klawiatury należy pamiętać o zarezerwowaniu w pamięci komputera odpowiednio dużego obszaru, w którym znaki te będą przechowywane.

```
char *Napis = new char [ 32 ] ;
for ( int Kolejny = 0 ; Kolejny < 10 ; ++Kolejny )
    *Napis++ = getche ( ) ;
```

Często popełnianym błędem jest zapisanie deklaracji zmiennej Napis jako

```
char *Napis ;
```

Wartość tej zmiennej jest w tym przypadku nieokreślona, czyli wskazuje przypadkowe miejsce pamięci. Znaki wprowadzane następnie za pomocą funkcji getche mogą więc niszczyć inne dane programu i doprowadzić do jego dezorganizacji. Wczytywane znaki można również umieścić w tablicy, pisząc

```
char Napis [ 12 ] ;
```

i używając indeksu do wskazywania kolejnych pozycji tablicy Napis.

Opisy funkcji bibliotecznych związanych z urządzeniami zewnętrznymi zawierają: postać definicji (typ wyniku, sygnatura), nazwę biblioteki, opis działania, przekazywany wynik i przykłady użycia funkcji.

### 9.1.1. getch

Deklaracja `int getch ( void ) ;`

Biblioteka conio.h

Działanie Funkcja getch odczytuje kolejny opis klawisza z bufora klawiatury – gdy bufor klawiatury jest pusty, to działanie programu jest zawieszane do czasu naciśnięcia klawisza znakowego lub sterującego.

Wynik Dla klawiszy znakowych wynikiem jest kod ASCII znaku związanego z tym klawiszem (z uwzględnieniem stanu klawiszy modyfikujących); dla klawiszy sterujących wynikiem jest 0 lub wartość 0xE0 – w tym przypadku po ponownym wywołaniu funkcji getch jej wynikiem jest numer techniczny naciśniętego klawisza.

Przykład

```
#include <conio.h>
void Test ( void )
{
    //.............
    int Znak, Sterowanie = 0 ;
    Znak = getch ( ) ;
    if ( Znak == 0 || Znak == 0xE0 )
        Sterowanie = getch ( ) ;
    //.............
}
```

### 9.1.2. getche

Deklaracja `int getche ( void ) ;`

Biblioteka `conio.h`

Działanie Funkcja getche działa tak samo jak funkcja getch, z tym że obraz odczytanego znaku jest dodatkowo wyświetlany na ekranie monitora (echo).

Wynik jak dla funkcji getch .

### 9.1.3. scanf

Deklaracja `int scanf ( const char *format, wskaźnik, wskaźnik, ... ) ;`

Biblioteka `stdio.h`

Działanie Funkcja scanf wczytuje kolejne pola (ciągi znaków), ograniczone znakiem spacji ' ' lub nowej linii '\n'. Obrazy wczytywanych znaków są wyświetlane na ekranie monitora. Liczba wczytywanych pól i sposób ich przetwarzania są zadawane za pomocą pierwszego argumentu funkcji scanf, oznaczonego identyfikatorem format, którego wartością jest ciąg znaków. Ciąg ten składa się z pewnej liczby wzorców konwersji – liczba wzorców występujących w formacie określa liczbę pól wczytywanych ze strumienia wejściowego i liczbę wskaźników będących dalszymi argumentami funkcji scanf . Wzorzec konwersji ma następującą postać:

% [ * ] [ *szerokość* ] [ *prefiks* ] *znak_konwersji*

Każdy wzorzec rozpoczyna się zawsze od znaku procenta %, składniki umieszczone w nawiasach prostokątnych nie muszą występować. Znak gwiazdki * użyty we wzorcu nakazuje pominąć kolejne pole znaków ze strumienia wejściowego. *Szerokość* określa maksymalną liczbę znaków, które mają być pobrane z kolejnego pola. Pole to może być krótsze niż zadana szerokość – w takim przypadku przetworzone będą jedynie znaki znajdujące się przed znakiem spacji lub nowej linii. *Prefiks* i *znak_konwersji* określają sposób przetworzenia znaków pola – wynikiem tego przetwarzania jest najczęściej wartość binarna liczby określonego typu. Kolejne wzorce konwersji są powiązane z kolejnymi wskaźnikami będącymi dalszymi argumentami funkcji scanf – obliczona wartość binarna liczby jest wprowadzana do obszaru pamięci wskazanego przez związany z danym wzorcem wskaźnik. Zakłada się przy tym, że typ wskaźnika związanego z wzorcem jest taki sam, jak typ konwersji zadanej w tym wzorcu. W tabelach podano znaczenie poszczególnych znaków konwersji i prefiksów

| znak k. | wejściowe pole znaków | typ wskaźnika |
|---|---|---|
| d | liczba całkowita dziesiętna | int* |
| D | liczba całkowita dziesiętna | long* |
| o | liczba całkowita oktalna | int* |
| O | liczba całkowita oktalna | long* |
| i | liczba całkowita dziesiętna, oktalna lub heksadecymalna | int* |
| l | liczba całkowita dziesiętna, oktalna lub heksadecymalna | long* |
| u | liczba całkowita dziesiętna bez znaku | unsigned int* |
| U | liczba całkowita dziesiętna bez znaku | unsigned long* |
| x | liczba całkowita heksadecymalna | int* |
| X | liczba całkowita heksadecymalna | long* |
| e, E | liczba zmiennopozycyjna | float* |
| f | liczba zmiennopozycyjna | float* |
| g, G | liczba zmiennopozycyjna | float* |
| s | ciąg znaków | char* |
| c | znak | char* |

| prefiks | znaki konwersji | typ wskaźnika |
|---|---|---|
| h | d, i, o, u, x | short* |
| l | d, i, o, u, x | long* |
| | e, f, g | double* |
| L | e, f, g | long double* |

Pola reprezentujące liczby (poza konwersją u, U ) mogą rozpoczynać się od znaków plus + lub minus − . W przypadku wczytywania ciągu znaków (konwersja s ) kody kolejnych znaków są lokowane w kolejnych bajtach pamięci, począwszy od bajtu wskazanego przez wskaźnik związany z danym wzorcem konwersji. Konieczne więc jest zarezerwowanie odpowiednio dużego obszaru pamięci, w którym znaki te będą umieszczane. Po ostatnim wczytanym znaku (różnym od znaku spacji lub nowej linii) jest umieszczany znak końca ciągu o wartości 0. Za pomocą konwersji s nie można wczytać tekstu złożonego z kilku słów oddzielonych spacjami. W takim przypadku można skorzystać z konwersji wyliczeniowej o postaci:

[ −~]

W nawiasach klamrowych podano przedział znaków ASCII (od znaku spacji do znaku tyldy), które będą akceptowane i wprowadzane. Pierwszy znak spoza tego przedziału (czyli np. znak nowej linii) kończy wprowadzany tekst. Podczas wprowadzania znaków z klawiatury można używać klawisza *Backspace* do kasowania poprzednio wprowadzonych znaków. *Prefiks* umieszczony przed znakiem konwersji redefiniuje znaczenie tego znaku.

. Wynikiem funkcji scanf jest liczba pól znaków, które zostały poprawnie wczytane, poddane konwersji, a obliczone wartości zostały zapamiętane

Przykład

```
#include <stdio.h>
void Test_1 ( void )
{
   // .............
   int alfa ;
   scanf ( "%d", &alfa ) ;
   // .............
}
void Test_2 ( void )
{
   char Znak ;
   int Element, Składnik, *Adres = &Składnik ;
   int Tablica [ 20 ] ;
   long Długi, Podłużny ;
   float Ułamek ;
   double Dokładny ;
   long double Precyzyjny ;
    char *Format = "%f%lf%Lf" ;
    char *Słowo = new char [ 10 ] ;
    char Znaki [ 16 ] ;
    char Tekst [ 65 ] ;
    // ...........
    scanf ( "%c%d", &Znak, &Element);
    scanf ( "%i", Adres ) ;
    scanf ( "%d%d", &Tablica [ 0 ], &Tablica [ 19 ] ) ;
    scanf ( "%D%ld", &Długi, &Podłużny ) ;
    scanf ( Format, &Ułamek, &Dokładny, &Precyzyjny) ;
    scanf( "%s%s", Słowo, Znaki ) ;
    scanf ( "%[ – ~]", Tekst ) ;
   // ..........
}
```

Uwaga Funkcja scanf przerywa działanie w przypadku napotkania znaku niedozwolonego (np. znaku litery w liczbie całkowitej). Znak ten nie jest usuwany ze strumienia wejściowego, co niekiedy powoduje zapętlenie programu. Sytuacji tej można uniknąć, sprawdzając liczbę przetworzonych poprawnie pól znakowych, przekazywaną jako wynik funkcji scanf. Przed wywołaniem funkcji scanf można też oczyścić bufor klawiatury za pomocą funkcji

```
fflush ( stdin ) ;
```

Operacja wyczyszczenia bufora jest szczególnie zalecana przed wczytywaniem pojedynczego znaku (konwersja *c*).

### 9.1.4. gets

Deklaracja `char* gets ( char *Tekst ) ;`

Biblioteka stdio.h

Działanie Funkcja gets odczytuje ciąg znaków, w którym mogą wystąpić znaki spacji lub znaki tabulacji poziomej. Odczytane znaki są zapamiętywane, począwszy od miejsca wskazanego przez argument Tekst. Pobieranie kolejnych znaków kończy wystąpienie znaku nowej linii – znak ten nie jest wpisywany na końcu zapamiętanego tekstu, zamiast niego wpisywany jest znak o kodzie 0 (koniec ciągu znaków). Odczytane znaki są równocześnie wyświetlane na ekranie monitora (echo).

Wynik Gdy wykonanie funkcji gets zakończyło się z powodu napotkania znaku nowej linii, to jej wynikiem jest wskaźnik zapamiętanego ciągu znaków (czyli argument Tekst )

Przykład

```
#include <stdio.h>
void Test ( void )
{
   .....
   char *Napis = new char [ 25 ] ;
   char Linia [ 16 ] ;
   gets ( Napis ) ;
   gets ( Linia ) ;
}
```

Uwaga Za pomocą funkcji gets można wprowadzić nieograniczoną liczbę znaków – może to być przyczyną uszkodzenia programu. Aby liczbę wprowadzanych znaków ograniczyć, można użyć dostępnej w niektórych bibliotekach funkcji gets_s, która umożliwia podanie jako drugiego argumentu maksymalnej liczby wczytywanych znaków.

## 9.2. Monitor ekranowy

W punkcie tym zostanie przedstawiony monochromatyczny tryb znakowy, będący podstawowym trybem wyświetlania informacji na monitorze ekranowym (w oknie konsoli). Wyświetlanie znaków na ekranie monitora realizowane jest za pomocą funkcji bibliotecznych. Kolejny znak wyprowadzany przez funkcję biblioteczną jest zawsze wyświetlany na pozycji wskazanej przez kursor, po czym współrzędne kursora są zmieniane tak, aby wskazywały następną z prawej pozycję w tym samym wierszu. Po osiągnięciu końca wiersza kursor przesuwany jest na początek następnego wiersza, a po wyświetleniu znaku w prawym, dolnym narożniku następuje przewinięcie ekranu.

### 9.2.1. putchar

Deklaracja `int putchar ( int Znak ) ;`

Biblioteka `stdio.h`

Działanie Funkcja putchar wyprowadza na ekran monitora znak określony za pomocą dwubajtowej liczby całkowitej. Młodszy bajt tej liczby to kod ASCII znaku, starszy bajt zawiera tryb wyświetlania znaku.

Wynik Gdy funkcja została wykonana poprawnie, wynikiem jest wyświetlany znak (czyli wartość argumentu Znak).

Przykład

```
#include <stdio.h>
void Test ( void )
{
    .............
    int Litera = 'a' ;
    putchar ( Litera ) ;
    .............
}
```

### 9.2.2. puts

Deklaracja `int puts ( char *Napis ) ;`

Biblioteka `stdio.h`

Działanie Funkcja puts wyprowadza na ekran ciąg znaków zadany przez wskaźnik Napis. Wyprowadzane są kolejne znaki tego ciągu, aż do napotkania znaku o kodzie równym 0, zamiast którego wyprowadzany jest znak nowej linii.

Wynik Gdy funkcja została wykonana poprawnie wynikiem jest wartość nieujemna.

Przykład

```
#include <stdio.h>
void  Test  ( void )
{
    //.............
    char *Tekst  =  "Napis ćwiczebny." ;
    puts ( Tekst ) ;
}
```

### 9.2.3. printf

Deklaracja `int printf (const char *format, wyrażenie, wyrażenie, ... ) ;`

Biblioteka `stdio.h`

Działanie Funkcja printf umożliwia wyprowadzanie ciągów znaków reprezentujących wartości wyrażeń, które są jej argumentami. Sposób wyprowadzania określa argument format będący ciągiem znaków zawierającym:

– znaki przesyłane bezpośrednio na ekran monitora,
– wzorce konwersji definiujące sposób przetwarzania wartości wyrażeń na ciągi znaków.

Liczba wzorców konwersji powinna być równa liczbie wyrażeń, kolejne wzorce określają sposób przetwarzania wartości kolejnych wyrażeń. Wzorzec konwersji ma postać:

% [ *opis* ] [ *szerokość* ] [ **.** *precyzja* ] [ *prefiks* ] *znak_konwersji*

Wzorzec konwersji rozpoczyna się od znaku procentu %, składniki zapisane w nawiasach kwadratowych mogą nie występować. *Prefiks* i *znak_konwersji* określają sposób przekształcania wartości wyrażenia odpowiadającego danemu wzorcowi konwersji na ciąg znaków – składniki te muszą być dostosowane do typu wartości wyrażenia.

| znak k. | typ wyrażenia | wynik konwersji |
|---|---|---|
| d | int | liczba dziesiętna ze znakiem |
| i | int | liczba dziesiętna ze znakiem |
| o | unsigned | liczba oktalna bez znaku |
| u | unsigned | liczba dziesiętna bez znaku |
| x | unsigned | liczba heksadecymalna bez znaku ( a ... f ) |
| X | unsigned | liczba heksadecymalna bez znaku ( A ... F) |

| | | |
|---|---|---|
| f | float | liczba rzeczywista ze znakiem o postaci ddd.ddd |
| e | float | liczba rzeczywista ze znakiem o postaci d.ddde[+/-]ddd |
| g | float | konwersja f lub e, zależnie od wartości argumentu |
| E | float | jak konwersja e z użyciem litery E do oznaczenia wykładnika |
| G | float | jak konwersja g z użyciem litery E do oznaczenia wykładnika |
| s | char* | ciąg znaków |
| c | char | pojedynczy znak |

| prefiks | znaki konwersji | typ wyrażenia |
|---|---|---|
| h | d, i, o, u, x, X | short |
| l | d, i, o, u, x, X | long |
| | e, E, f, g, G | double |
| L | e, E, f, g, G | long double |

W przypadku konwersji s, przeznaczonej dla ciągów znaków, wyprowadzanych jest co najwyżej tyle znaków, ile wskazano za pomocą składnika *szerokość*. Gdy składnik ten nie występuje we wzorcu, wyprowadzane są wszystkie znaki ciągu. Znak nowej linii '\n' jest zamieniany przez funkcję printf na parę znaków CR i LF, co powoduje przesunięcie kursora na początek następnego wiersza (z ewentualnym przewinięciem ekranu). Poprawnie są interpretowane również znaki sterujące: '\t' (tabulacja pozioma HT ), '\r' (powrót kursora CR ), '\b' (cofnięcie kursora ze zmazaniem poprzedniego znaku BS ) i '\a' (dzwonek BELL ). Składnik *szerokość* określa minimalną liczbę znaków, które zostaną wyprowadzone. Większa liczba znaków zostanie wyprowadzona, gdy w wyniku konwersji otrzymano ciąg znaków dłuższy od podanej szerokości. Jeżeli natomiast wynik konwersji zawiera mniej znaków, to zostanie on uzupełniony znakami spacji. Składnik *opis* jest jednym z trzech znaków:

– uzupełnianie znakami spacji z prawej strony; gdy opis nie występuje, uzupełnia się z lewej strony,

\+ wyprowadzanie znaku liczby (plus albo minus); gdy opis nie występuje, wyprowadzany jest jedynie minus,

**spacja** wyprowadzanie znaku spacji zamiast znaku plus liczby.

Składnik *precyzja* rozpoczyna się od znaku kropki i jest wartością całkowitą określającą liczbę miejsc po kropce dziesiętnej wyprowadzanej wartości zmiennopozycyjnej lub liczbę znaków podczas wyprowadzania ciągu znaków (konwersja s). Gdy precyzja nie jest określona, wyprowadzanych jest 6 miejsc po kropce dziesiętnej.

Wynik Wynikiem funkcji printf jest łączna liczba wyprowadzonych znaków.

Przykład

```
#include  <stdio.h>
void Test_1 ( void )
{
     //.............
     int  Rezultat = 155 ;
     printf ( "Wynik = %d\n",  Rezultat ) ;
    //.............
}

void Test_2 ( void )
{
     //.............
     int   Bok  =  15 ;
     printf ( "Wartość zmiennej Bok wynosi %d",  Bok ) ;
     printf( "%d",  Bok  *  Bok  + 2 ) ;
     int  Szyk [ 5 ]  =  { 1, 2, 3, 4, 5 } ;
     for ( int  Kolejny  =  0 ;  Kolejny < 5 ;  ++Kolejny )
       printf( "\nSzyk [%d]=%d", Kolejny, Szyk [ Kolejny ] ) ;
     //
     float  Skok  =  151.37825;
     printf ( "%+12.3f",  Skok ) ;       //   +151.378
     printf ( "%E", sin ( 1 – 1 / Skok ) * cos ( 1 / exp( Skok ) ) ) ;
     //
     long  Czas  =  3782505 ;
     double  Księżyc  =  27E38 ;
     long double  Mikrus  =  –5E–127 ;
     printf ( "%ld \t %le \t %Le" , Czas,  Księżyc,  Mikrus ) ;
     //
     char  *Napis  =  "raz dwa trzy" ;
     printf( "%s",  Napis ) ;              // wszystkie znaki
     printf ( "%.7s",  Napis ) ;          // 7 początkowych znaków
     printf ( "\n" ) ;                    // nowa linia
      //.........
}
```

## 9.3. Pliki dyskowe

W punkcie tym zostanie przedstawiony zestaw funkcji bibliotecznych umożliwiających korzystanie z sekwencyjnych plików dyskowych. Plik dyskowy przetwarzany za pomocą tych funkcji może być uważany za ciąg pozycji bajtowych, w którym wyróżniona jest jedna pozycja aktualna. Zapis lub odczyt danych rozpoczyna się zawsze od pozycji aktualnej i powoduje przesunięcie tej pozycji w kierunku końca pliku o liczbę przeczytanych lub zapisanych bajtów. Podczas zapisu danych, po osiągnięciu końca pliku, długość ciągu bajtów reprezentujących ten plik jest automatycznie zwiększana. Korzystanie z pliku rozpoczyna się od jego otwarcia, czyli od zgłoszenia systemowi operacyjnemu potrzeby dostępu do wskazanego pliku dyskowego. Operacja otwarcia pliku wytwarza wskaźnik pliku, który powinien stać się wartością zmiennej wskaźnikowej typu FILE*. Typ ten jest zdefiniowany w bibliotece stdio.h. Po wykonaniu wszystkich operacji zapisu lub odczytu danych, plik powinien zostać zamknięty. Wśród funkcji standardowych umożliwiających korzystanie z sekwencyjnych plików dyskowych można wyróżnić 3 grupy:

- funkcje fopen, fclose otwierające i zamykające pliki,
- funkcje fseek, rewind, feof ustalające aktualną pozycję pliku i badające osiągnięcie końca pliku,
- funkcje fgetc, fputc, fscanf, fprintf, fread, fwrite czytające i zapisujące dane.

### 9.3.1. fopen

| | |
|---|---|
| Deklaracja | `FILE* fopen ( const char *Nazwa, const char *Tryb ) ;` |
| Biblioteka | stdio.h |
| Działanie | Funkcja fopen otwiera plik wskazany za pomocą argumentu Nazwa – może to być nazwa pliku odnosząca się do katalogu bieżącego lub określenie pełnej ścieżki dostępu do pliku. Drugi argument Tryb ustala sposób dostępu do pliku: |

r odczyt istniejącego pliku,

w utworzenie pliku do zapisu,

a zapis na końcu istniejącego pliku,

r+ zapis lub odczyt istniejącego pliku,

w+ utworzenie pliku do zapisu i odczytu,

a+ zapis lub odczyt na końcu istniejącego pliku.

Po otwarciu aktualną pozycją jest początkowy bajt pliku dla trybów r, r+ lub w , w+ albo bajt znajdujący się za ostatnim dotąd zapisanym bajtem dla trybów a, a+. Otwarcie istniejącego już pliku w trybie w lub w+ powoduje utratę zapisanych w nim dotąd danych. Gdy natomiast plik otwierany w trybie a lub a+ jeszcze nie istnieje, to zostanie on utworzony. Wymienione tryby można uzupełnić literą t lub b (np. rt, rt+, wb, a+b ). Litera t oznacza plik tekstowy, litera b plik binarny. Jeżeli plik został otwarty w celu aktualizacji ( + ), to można dokonywać odczytu i zapisu danych z/do tego pliku. Przy zmianie rodzaju dostępu (zapis – odczyt, odczyt – zapis) konieczne jest jednak wykonanie funkcji ustalającej aktualną pozycję pliku (fseek).

Wynik Gdy plik został otwarty, wynikiem funkcji fopen jest wskaźnik pliku, po nieudanej próbie otwarcia pliku wynikiem jest NULL.

Przykład

```
#include <stdio.h>
void  Test ( void )
{
    //.....
    FILE  *SpisTowarów ;
    if ( ( SpisTowarów = fopen( "SPIS.TXT" , "rt+" ) ) == NULL )
         printf ( " \nNie można otworzyć pliku." ) ;
   //.....
}
```

### 9.3.2. fclose

Deklaracja `int  fclose ( FILE  *Plik ) ;`

Biblioteka stdio.h

Działanie Funkcja fclose zamyka plik, którego wskaźnik jest wartością argumentu Plik. Po zamknięciu pliku nie można wykonywać żadnych operacji dotyczących tego pliku (z wyjątkiem ponownego otwarcia).

Wynik Wynikiem funkcji fclose jest 0 , gdy operacja zamknięcia pliku została zakończona poprawnie, w przeciwnym wypadku wynikiem jest EOF.

Przykład

```
#include <stdio.h>
void  Test ( void )
{
    FILE  *Personel  =  fopen ( "PER.TXT" , "rt" ) ;
    // .....
    fclose ( Personel ) ;
}
```

### 9.3.3. fseek

Deklaracja `int fseek ( FILE *Plik, long Pozycja, int Start ) ;`

Biblioteka `stdio.h`

Działanie Funkcja fseek ustala aktualną pozycję pliku wskazanego za pomocą argumentu Plik. Wartością argumentu Pozycja powinna być liczba określająca numer bajtu pliku, liczony od miejsca wskazanego przez argument Start. Argumentem Start może być jedna ze stałych:

| | |
|---|---|
| SEEK_SET | początek pliku, |
| SEEK_CUR | aktualna pozycja pliku, |
| SEEK_END | koniec pliku. |

Wynik Wynikiem funkcji fseek jest 0, gdy aktualna pozycja pliku została poprawnie ustalona – w przeciwnym wypadku wynikiem jest wartość niezerowa.

Przykład

```
#include <stdio.h>
void  Test ( void )
{
   //.....
   FILE  *Opis  =  fopen ( "OPIS.DOC", "rt+" ) ;
   fseek ( Opis, 0L, SEEK_END ) ;
   // .....                              // zapis na końcu pliku
   fseek ( Opis, 0L, SEEK_SET ) ;
   //.....                              // zapis na początku pliku
   fseek ( Opis,  1500L, SEEK_CUR ) ;
   //.....                              // odczyt z pliku
}
```

### 9.3.4. feof

Deklaracja `int feof ( FILE  *Plik ) ;`

Biblioteka `stdio.h`

Działanie Funkcja feof odczytuje stan znacznika końca pliku.

Wynik Gdy podczas ostatniej operacji odczytu napotkano koniec pliku, to wynikiem funkcji feof jest wartość niezerowa, w przeciwnym wypadku wynikiem jest 0 .

Przykład

```
#include <stdio.h>
void Test ( void )
{
    //.....
    FILE *PlikRoboczy = fopen ( "ROBOTA.MAN", "rt" ) ;
   // .....                    // odczyt z pliku

   if ( feof ( PlikRoboczy ) !=0 )
       printf ( "\nKoniec pliku." ) ;
}
```

### 9.3.5. fgetc

Deklaracja `int fgetc ( FILE *Plik ) ;`

Biblioteka stdio.h

Działanie Funkcja fgetc odczytuje ze wskazanego pliku kolejny znak.

Wynik Wynikiem funkcji fgetc jest liczba całkowita, której młodszy bajt to wartość odczytanego znaku, a starszy bajt równy jest 0 – gdy wystąpił błąd, wynikiem jest EOF.

Przykład

```
#include <stdio.h>
void Test ( void )
{
// .....
   char Znak ;
   FILE *DanePomocnicze = fopen ( "DANE.DOC" , "rt" ) ;
   Znak = fgetc ( DanePomocnicze ) ;
 // .....
}
```

### 9.3.6. fputc

Deklaracja `int fputc ( int Znak, FILE *Plik ) ;`

Biblioteka stdio.h

Działanie Funkcja fputc zapisuje do wskazanego pliku kolejny znak zadany za pomocą liczby całkowitej – młodszy bajt powinien zawierać kod znaku.

Wynik Wynikiem funkcji fputc jest zapisany znak, a w przypadku błędu zapisu wynikiem jest EOF.

Przykład

```
#include <stdio.h>
void Test ( void )
{
    char Posuw = 'K' ;
    FILE *OpisTokarki = fopen ( "TOKARKA.DOC" , "rt+" ) ;
    //.....
    fputc ( Posuw, OpisTokarki ) ;
}
```

### 9.3.7. fscanf

Deklaracja

```
int fscanf ( FILE *Plik, const char *format,
                                    wskaźnik, wskaźnik, ... ) ;
```

Biblioteka stdio.h

Działanie Funkcja fscanf odczytuje ze wskazanego pliku ciągi znaków, dokonuje ich konwersji na wartości binarne i zapamiętuje te wartości w miejscach pamięci wskazanych za pomocą argumentów będących wskaźnikami. Liczba wczytywanych ciągów znaków i sposób ich przetwarzania zadawane są argumentem format, którego wartością jest ciąg znaków złożony z wzorców konwersji. Postać wzorców konwersji jest taka sama jak dla funkcji scanf.

Wynik Wynikiem funkcji fscanf jest liczba ciągów znaków, które zostały poprawnie odczytane, przetworzone na wartości binarne, a uzyskane wartości są zapisane w pamięci komputera. W przypadku napotkania końca pliku wynikiem jest EOF .

Przykład

```
#include <stdio.h>
void Test ( void )
{   //.....
    int Sztuki ;
    float Cena ;
    FILE *Towar = fopen ( "SPIS_TOWAROW.DOC" , "rt+" ) ;
    //.....
    fscanf ( Towar, "%d%f", &Sztuki, &Cena ) ;
    //.....
}
```

### 9.3.8. fprintf

Deklaracja
```
int fprintf ( FILE *Plik, const char *format,
                                    wyrażenie, wyrażenie, ... ) ;
```

Biblioteka stdio.h

Działanie Funkcja fprintf zapisuje we wskazanym pliku ciągi znaków zadane za pomocą wyrażeń będących jej argumentami. Sposób konwersji wartości wyrażeń na ciągi znaków określa argument format, będący ciągiem znaków zawierającym znaki wpisywane bezpośrednio do pliku dyskowego i wzorce konwersji. Postać wzorca konwersji jest taka sama jak dla funkcji printf .

Wynik Wynikiem funkcji fprintf jest liczba zapisanych bajtów, a w przypadku wystąpienia błędu wynikiem jest EOF.

Przykład
```
#include <stdio.h>
void Test ( void )
{
    .....
    int KodWaluty = 178 ;
    float KursBieżący = 0.39 ;
    FILE *TabelaKursów = fopen ( "KURSY.TAB" , "rt+" ) ;
    //.....
    fprintf ( TabelaKursów, "\n%3d\t%8.3f",
                            KodWaluty, KursBieżący ) ;
    //.....
}
```

### 9.3.9. fread

Definicja
```
int fread ( wskaźnik, int Rozmiar, int Liczba, FILE *Plik ) ;
```

Biblioteka stdio.h

Działanie Funkcja fread odczytuje ze wskazanego pliku dyskowego zadaną liczbę struktur danych; każda z nich powinna mieć długość określoną za pomocą argumentu Rozmiar. Odczytane dane są umieszczane w miejscu pamięci określonym przez pierwszy argument (wskaźnik).

Wynik Wynikiem funkcji fread jest liczba odczytanych struktur. W przypadku napotkania końca pliku wynikiem jest EOF.

Przykład

```
#include  <stdio.h>
void  Test ( void )
{    //.....
     struct książka
     {
          char  Autor [ 25 ] ;
          char  Tytuł [ 50 ] ;
     }  Książki [ 100 ] ;

     //.....
     FILE  *Magazyn  =  fopen ( "MAGAZYN.DOC" , "rt+" ) ;
     //.....
     fread ( Książki,  sizeof ( książka ), 100, Magazyn ) ;
     //.....
}
```

### 9.3.10. fwrite

Deklaracja `int fwrite ( wskaźnik, int Rozmiar, int Liczba, FILE *Plik ) ;`

Biblioteka `stdio.h`

Działanie Funkcja fwrite zapisuje w pliku dyskowym wskazaną liczbę struktur danych; każda z nich powinna mieć długość określoną za pomocą argumentu Rozmiar. Zapisywane dane są pobierane z miejsca pamięci określonego przez pierwszy argument (wskaźnik).

Wynik Wynikiem funkcji fwrite jest liczba zapisanych struktur. W przypadku wystąpienia błędu wynikiem jest EOF.

Przykład

```
#include  <stdio.h>
void  Test ( void )
{
    //.....
    double  Pomiary [ Wierszy ] [ Kolumn ] ;
    //.....
    FILE  *Archiwum  =  fopen ( "ARCHIWUM.TAB" , "w+" ) ;
    fwrite (Pomiary,  sizeof ( Pomiary ), 1, Archiwum ) ;
    //.....
}
```

# DODATKI

## Dodatek A

## Kod ASCII

| Znak | D | H | Znak | D | H | Znak | D | H | Znak | D | H |
|---|---|---|---|---|---|---|---|---|---|---|---|
| NUL | 00 | 00 | DLE | 16 | 10 | SP | 32 | 20 | 0 | 48 | 30 |
| SOH | 01 | 01 | DC1 | 17 | 11 | ! | 33 | 21 | 1 | 49 | 31 |
| STX | 02 | 02 | DC2 | 18 | 12 | " | 34 | 22 | 2 | 50 | 32 |
| ETX | 03 | 03 | DC3 | 19 | 13 | # | 35 | 23 | 3 | 51 | 33 |
| EOT | 04 | 04 | DC4 | 20 | 14 | $ | 36 | 24 | 4 | 52 | 34 |
| ENQ | 05 | 05 | NAK | 21 | 15 | % | 37 | 25 | 5 | 53 | 35 |
| ACK | 06 | 06 | SYN | 22 | 16 | & | 38 | 26 | 6 | 54 | 36 |
| BEL | 07 | 07 | ETB | 23 | 17 | ' | 39 | 27 | 7 | 55 | 37 |
| BS | 08 | 08 | CAN | 24 | 18 | ( | 40 | 28 | 8 | 56 | 38 |
| HT | 09 | 09 | EM | 25 | 19 | ) | 41 | 29 | 9 | 57 | 39 |
| LF | 10 | 0A | SUB | 26 | 1A | * | 42 | 2A | : | 58 | 3A |
| VT | 11 | 0B | ESC | 27 | 1B | + | 43 | 2B | ; | 59 | 3B |
| FF | 12 | 0C | FS | 28 | 1C | , | 44 | 2C | < | 60 | 3C |
| CR | 13 | 0D | GS | 29 | 1D | – | 45 | 2D | = | 61 | 3D |
| SO | 14 | 0E | RS | 30 | 1E | . | 46 | 2E | > | 62 | 3E |
| SI | 15 | 0F | US | 31 | 1F | / | 47 | 2F | ? | 63 | 3F |

| Znak | D | H | Znak | D | H | Znak | D | H | Znak | D | H |
|---|---|---|---|---|---|---|---|---|---|---|---|
| @ | 64 | 40 | P | 80 | 50 | ‘ | 96 | 60 | p | 112 | 70 |
| A | 65 | 41 | Q | 81 | 51 | a | 97 | 61 | q | 113 | 71 |
| B | 66 | 42 | R | 82 | 52 | b | 98 | 62 | r | 114 | 72 |
| C | 67 | 43 | S | 83 | 53 | c | 99 | 63 | s | 115 | 73 |
| D | 68 | 44 | T | 84 | 54 | d | 100 | 64 | t | 116 | 74 |
| E | 69 | 45 | U | 85 | 55 | e | 101 | 65 | u | 117 | 75 |
| F | 70 | 46 | V | 86 | 56 | f | 102 | 66 | v | 118 | 76 |
| G | 71 | 47 | W | 87 | 57 | g | 103 | 67 | w | 119 | 77 |
| H | 72 | 48 | X | 88 | 58 | h | 104 | 68 | x | 120 | 78 |
| I | 73 | 49 | Y | 89 | 59 | i | 105 | 69 | y | 121 | 79 |
| J | 74 | 4A | Z | 90 | 5A | j | 106 | 6A | z | 122 | 7A |
| K | 75 | 4B | [ | 91 | 5B | k | 107 | 6B | { | 123 | 7B |
| L | 76 | 4C | \ | 92 | 5C | l | 108 | 6C | \| | 124 | 7C |
| M | 77 | 4D | ] | 93 | 5D | m | 109 | 6D | } | 125 | 7D |
| N | 78 | 4E | ^ | 94 | 5E | n | 110 | 6E | ~ | 126 | 7E |
| O | 79 | 4F | _ | 95 | 5F | o | 111 | 6F | DEL | 127 | 7F |

# Dodatek B

## Numery techniczne klawiszy

### Klawisze edycyjne

| Klawisz | D | H | Klawisz | D | H |
|---|---|---|---|---|---|
| Print Screen | 55 | 37 | Ins | 82 | 52 |
| Scroll Lock | 70 | 46 | Home | 71 | 47 |
| Pause | 69 | 45 | Page Up | 73 | 49 |
| ↑ | 72 | 48 | Del | 83 | 53 |
| ← | 75 | 4B | End | 79 | 4F |
| ↓ | 80 | 50 | Page Down | 81 | 51 |
| → | 77 | 4D | | | |

### Klawisze funkcyjne

| Klawisz | D | H | Klawisz | D | H |
|---|---|---|---|---|---|
| F1 | 59 | 3B | F7 | 65 | 41 |
| F2 | 60 | 3C | F8 | 66 | 42 |
| F3 | 61 | 3D | F9 | 67 | 43 |
| F4 | 62 | 3E | F10 | 68 | 44 |
| F5 | 63 | 3F | F11 | 87 | 57 |
| F6 | 64 | 40 | F12 | 88 | 58 |

## Klawisze numeryczne

| Klawisz | D | H | Klawisz | D | H |
|---|---|---|---|---|---|
| 7 Home | 71 | 47 | 0 Ins | 82 | 52 |
| 8 ↑ | 72 | 48 | . Del | 83 | 53 |
| 9 PgUp | 73 | 49 | NumLock | 69 | 45 |
| 4 ← | 75 | 4B | / | 53 | 35 |
| 5 | 76 | 4C | * | 55 | 37 |
| 6 → | 77 | 4D | – | 74 | 4A |
| 1 End | 79 | 4F | + | 78 | 4E |
| 2 ↓ | 80 | 50 | Enter | 28 | 1C |
| 3 PgDn | 81 | 51 | | | |

## Klawisze podstawowe

| Klawisz | D | H | Klawisz | D | H | Klawisz | D | H |
|---|---|---|---|---|---|---|---|---|
| Esc | 01 | 01 | Y | 21 | 15 | ` ~ | 41 | 29 |
| 1 ! | 02 | 02 | U | 22 | 16 | Shift (L) | 42 | 2A |
| 2 @ | 03 | 03 | I | 23 | 17 | \ \| | 43 | 2B |
| 3 # | 04 | 04 | O | 24 | 18 | Z | 44 | 2C |
| 4 $ | 05 | 05 | P | 25 | 19 | X | 45 | 2D |
| 5 % | 06 | 06 | [ { | 26 | 1A | C | 46 | 2E |
| 6 ^ | 07 | 07 | ] } | 27 | 1B | V | 47 | 2F |
| 7 & | 08 | 08 | Enter | 28 | 1C | B | 48 | 30 |
| 8 * | 09 | 09 | Ctrl | 29 | 1D | N | 49 | 31 |
| 9 ( | 10 | 0A | A | 30 | 1E | M | 50 | 32 |
| 0 ) | 11 | 0B | S | 31 | 1F | , < | 51 | 33 |
| – _ | 12 | 0C | D | 32 | 20 | . > | 52 | 34 |
| = + | 13 | 0D | F | 33 | 21 | / ? | 53 | 35 |
| Backspace | 14 | 0E | G | 34 | 22 | Shift (P) | 54 | 36 |
| Tab | 15 | 0F | H | 35 | 23 | Alt | 56 | 38 |
| Q | 16 | 10 | J | 36 | 24 | Space | 57 | 39 |
| W | 17 | 11 | K | 37 | 25 | CapsLock | 58 | 3A |
| E | 18 | 12 | L | 38 | 26 | | | |
| R | 19 | 13 | ; : | 39 | 27 | | | |
| T | 20 | 14 | ' " | 40 | 28 | | | |

# Dodatek C

## Słowa kluczowe

| | | | |
|---|---|---|---|
| auto | else | main | template |
| | enum | | this |
| bool | extern | operator | true |
| break | | | try |
| | false | private | typedef |
| catch | float | protected | |
| case | for | public | union |
| char | friend | | unsigned |
| class | | register | |
| const | if | return | virtual |
| continue | inline | | void |
| | int | short | |
| default | | signed | while |
| delete | long | sizeof | |
| do | | static | |
| double | new | struct | |
| | | switch | |

# Dodatek D

## Priorytety i łączność operatorów

| Operator | Priorytet | Łączność | Działanie |
|---|---|---|---|
| :: | 17 | P | zasięg |
| :: | | L | globalizacja |
| . | 16 | L | dostęp do składowej |
| –> | | L | dostęp do składowej |
| [ ] | | L | indeksowanie |
| ( ) | | L | wywołanie funkcji |
| sizeof | 15 | L | rozmiar typu |
| ++ | | P | postfiksowe zwiększanie o 1 |
| ++ | | P | prefiksowe zwiększanie o 1 |
| – – | | P | postfiksowe zmniejszanie o 1 |
| – – | | P | prefiksowe zmniejszanie o 1 |
| ~ | | P | negacja bitowa |
| ! | | P | negacja logiczna |
| – | | P | minus jednoargumentowy |
| + | | P | plus jednoargumentowy |
| & | | P | adres argumentu |

| Operator | Priorytet | Łączność | Działanie |
|---|---|---|---|
| \|\| | 4 | L | alternatywa logiczna |
| ? : | 3 | L | wyrażenie warunkowe |
| = | 2 | P | przypisanie |
| *= | | P | mnożenie i przypisanie |
| /= | | P | dzielenie i przypisanie |
| %= | | P | modulo i przypisanie |
| += | | P | dodawanie i przypisanie |
| −= | | P | odejmowanie i przypisanie |
| <<= | | P | przesunięcie w lewo i przypisanie |
| >>= | | P | przesunięcie w prawo i przypisanie |
| &= | | P | koniunkcja bitowa i przypisanie |
| ^= | | P | różnica symetryczna i przypisanie |
| \|= | | P | alternatywa bitowa i przypisanie |
| , | 1 | L | ustalenie kolejności |

Priorytet: im większa wartość, tym wyższy priorytet
Łączność: L – lewostronna, P – prawostronna