# Język C - podstawy programowania

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf”.
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

**Człowiek-najlepsza inwestycja**

UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
WYDZIAŁ MATEMATYKI, FIZYKI I INFORMATYKI
INSTYTUT INFORMATYKI

# Język C – podstawy programowania

Paweł Mikołajczak


LUBLIN 2011

Paweł Mikołajczak

JĘZYK C – PODSTAWY PROGRAMOWANIA

**Recenzent**: Marek Stabrowski
Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach
Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl

**Wydawca**
Instytut Informatyki
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

**Druk**
ESUS Agencja Reklamowo-Wydawnicza Tomasz Przybylak
ul. Ratajczaka 26/8
61-815 Poznań
www: www.esus.pl

ISBN: 978-83-62773-10-7

# SPIS TREŚCI

# PRZEDMOWA

Język C zajmuje wyjątkową pozycję wśród języków programowania. Od ponad czterdziestu lat jest ciągle najpopularniejszym językiem wśród zawodowych programistów. Ta wyróżniona pozycja języka C jest spowodowana faktem, że w zasadzie jest to język umiejscowiony pomiędzy językami niskiego poziomu (np. asemblerowymi) i językami wysokiego poziomu (np. Pascal, Java czy LISP). Jest wiele powodów popularności języka C, efektywność jest jego główną zaletą. Program komputerowy napisany w języku C i skompilowany będzie prawie tak samo szybki jak program, który zostanie napisany w asemblerze. Język C został opracowany w 1972 roku przez Denisa Ritchie'go oraz Kena Thompsona na bazie języka B, tworzonego przez Thompsona podczas jego prac na systemem operacyjnym UNIX. Ponieważ język C był tworzony, jako narzędzie pracy programistów, jego główną cechą jest elastyczność i użyteczność. Przyglądając się cechom współczesnych języków programowania (np. C++, Java, Phyton) widzimy ich duże podobieństwo do języka C. Dzięki znajomości języka C, programista o wiele łatwiej nauczy się nowego języka programowania. Język C jest językiem o wielu zaletach. Podstawowe cechy to wszechstronność (udostępnia wszystkie techniki i funkcje sterujące, które są potrzebne w praktyce programowania). Programy napisane w języku C mają niewielką objętość i charakteryzują się jedną z największych szybkości działania. Kompilatory języka C mogą być wykorzystane na ponad czterdziestu platformach systemowych. Moc i elastyczność języka C jest tak duża, że jest to podstawowe narzędzie do pisania systemów operacyjnych (np. UNIX, Linux, Windows). Może to być niespodzianką dla czytelnika, ale wiele znanych kompilatorów zostało napisanych za pomocą języka C (np. FORTRAN, APL, Pascal, LISP, BASIC) . Obecnie silnie rozwijana przez firmę NVIDIA technologia CUDA wykorzystuje w zasadzie język C w programowaniu kart graficznych. Ponieważ język C pozwala na wykonywanie operacji niskopoziomowych (np. manipulowanie bitami w bajtach), oferuje dostęp do sprzętu, dzięki czemu mamy możliwość programowania komputerów wchodzących w skład systemów wbudowanych.

Oczywiście, język C posiada także wady. W wielu opracowaniach wskazuje się, że wskaźniki są niebezpieczne, wykorzystując je musimy być odpowiedzialni i nieustannie czujni. Podobnie pewne kłopoty może sprawiać brak kontroli indeksów w tablicach, ponieważ w języku nie ma wbudowanych mechanizmów kontroli. Programista ma dużą swobodę, ale także na nim spoczywa duża odpowiedzialność. Pierwszym przyjętym standardem języka był standard określany, jako K&R albo klasyczne C. Podstawą przyjętego standardu

była opublikowana w 1978 roku przez Braina Kernighana i Dennisa Ritchie'go monografia pt. „ The C Programming Language" z dodatkiem pt. „ C Reference Manual". W roku 1983 Amerykański Narodowy Instytut Standardów (ANSI) powołał komitet w celu opracowania kolejnego standardu języka C. W 1989 standard został zatwierdzony i jest znany, jako standard ANSI C. W 1990 roku Międzynarodowa Organizacja Standaryzacji ISO zatwierdziła nowy kolejny standard, zwany standardem ISO 90. Standardy ISO C i ANSI C są w zasadzie tymi samymi standardami i funkcjonują pod wspólną nazwą, jako standard C90 (aczkolwiek można też spotkać nazwę C89). Najnowszy standard zatwierdzony przez wymienione dwie organizacje w 1999 roku, jako ANSI C99 i ISO/IEC 9899 funkcjonuje, jako standard C99.

Niniejszy podręcznik jest przeznaczony głównie dla studentów 3-letnich studiów zawodowych (licencjatów) a także dla studentów 2-letnich studiów uzupełniających. Podręcznik powstał na podstawie notatek wykorzystywanych przez Autora w trakcie prowadzenia wykładów z przedmiotów „Języki programowanie", oraz „Język C i C++" dla studentów UMCS w latach 1995 – 2007 i dalszych.

W podręczniku przyjęto zasadę, że nauka programowania oparta jest na licznych przykładach, które ilustrują praktyczne zastosowania paradygmatów programowania i składni języka C.

Podręcznik ma być pomocą dydaktyczną wspierającą naukę programowania realizowaną w ramach 30-godzinnego wykładu i ćwiczeń laboratoryjnych.

Należy pamiętać, że podręcznik nie stanowi pełnego kompendium wiedzy o języku C, jest jedynie prostym i przystępnym wprowadzeniem w fascynujący świat pisania efektywnych i efektownych programów komputerowych. Do pełnego opisu wszystkich niuansów języka C potrzeba obszernego podręcznika (np. podręcznik „Język C, szkoła programowania" Stephena Prata liczy sobie 743 strony).

Wszystkie przykłady programów, które zostały zamieszczone w tym podręczniku zostały przetestowane za pomocą kompilatora C++ Builder 6 firmy Borland, na platformie Windows. Większość programów sprawdzana była także na platformie Linuksowej z kompilatorem GCC.

# ROZDZIAŁ 1

## WPROWADZENIE

## 1.1. Programowanie

Za pomocą komputerów rozwiązujemy problemy (zadania). Problemy mogą być najróżniejsze, najprościej jest, jako przykłady rozważać problemy matematyczne. Na przykład, jeżeli mamy listę S, rozumianą, jako zestaw n liczb (elementów) ułożonych w określonym ciągu, to naszym zadaniem do rozwiązania jest posortowanie listy S w porządku niemalejącym. Niech S ma postać:

[ 15, 24, 7, 18 ]

Rozwiązaniem zadania jest następujący ciąg:

[ 7, 15, 18, 24 ]

Rozwiązanie zadania możemy znaleźć sami, przyglądając się elementom listy S i sortując je w pamięci, możemy też zlecić to zadanie do wykonania komputerowi. W każdym przypadku musimy wykonać sekwencję specjalnych działań:

- Określić **metodę**, która może być zastosowana
- Ustalić **plan** stosowania tej metody do konkretnego zadania
- Ustalić **opis czynności** wykonywanych podczas realizacji tego planu wraz z opisem ich skutków
- Otrzymać ostateczny **wynik** wykonywanych czynności.

Rozwiązanie zadania formułujemy, jako ciąg kroków, które należy wykonać. W informatyce najważniejsze są plany rozwiązywania zadań, ponieważ wykonanie planów powierzamy komputerom. Opis planu rozwiązywania zadania nosi nazwę **algorytmu.** Algorytmy składają się ze starannie dobranych **instrukcji elementarnych**, które są zleceniami wykonania akcji podstawowych. Rozważmy następujący problem: w zdefiniowanym ciągu S należy określić czy konkretna liczba x znajduje się na liście S. Możemy na przykład zapytać, czy w liście S występuje liczba x = 13. Istnieje wiele rozwiązań tego zadania, intuicyjnie najprostsze rozwiązanie może mieć następującą postać. Bierzemy pierwszy element listy S i porównujemy jego wartość z wartością x. Jeżeli są równe twierdzimy, że x jest elementem listy, jeżeli wartości nie są równe bierzemy kolejny element listy i znowu porównujemy go z wartością x. Postępujemy w ten sposób tak długo aż znajdziemy w S element x albo wyczerpiemy elementy listy. Przeglądając przykładowy zbiór stwierdzamy, że element x = 13 nie jest elementem listy S. Opisana przez nas procedura poszukiwania rozwiązania jest **algorytmem** rozwiązującym nasze zadanie.

Komputer może zostać wykorzystany do rozwiązywania zadania. W tym celu musimy dostarczyć komputerowi odpowiedni zestaw instrukcji, które opisują, jakie czynności muszą być wykonane.

Zestaw tych czynności nosi nazwę **programu komputerowego**, a pisanie i testowanie programów komputerowych nazywa się **programowaniem**.

Aby zakodować algorytm na użytek komputera, posługujemy się **językiem programowania.** Programowanie polega na zapisywaniu algorytmów w postaci programów komputerowych**.** Istnieje wiele formalnych i ogólnych definicji programowania, klasyczna definicja ma postać (A. i K. Kingsley-Hughes):

> *Programowanie to zdolność komunikacji z komputerami w zrozumiałym dla nich języku, z wykorzystaniem gramatyki i składni w celu wykonania użytecznych działań.*

To, co pisze programista jest nazywane **kodem** (a bardziej precyzyjnie – **kodem źródłowym**). Rozwiązywanie problemów za pomocą komputera nieodłącznie prowadzi do stworzenia odpowiednich reguł postępowania, inaczej mówiąc - algorytmu. Znaczenie nazwy algorytm jest bliskie takim znaczeniom jak przepis, metoda, sposób rozwiązania. W informatyce, algorytm ma oczywiście bardzo precyzyjne znaczenie. **Instrukcje**, dzięki którym za pomocą komputera rozwiązywane jest zadania, są bardziej precyzyjne, niż np. instrukcje, które mówią kucharzowi jak upiec ciastko, a cykliście jak złożyć rower.

Proces rozwiązywania problemów (zadań) za pomocą komputera składa się z dwóch odrębnych części:

- Analiza lub definiowanie problemu (zadania)
- Projektowanie i implementowanie algorytmu rozwiązującego zadanie

**Implementacja** oznacza produkt finalny, czyli zrozumiały dla konkretnego komputera zakodowany ciąg instrukcji.

Rozważania o algorytmach zaczniemy od analizy prostego stwierdzenia:

*"człowiek reaguje na środowisko"*

Zastanówmy się, co to znaczy "reaguje"? Z otaczającego nas świata nieustannie odbieramy różnego typu informacje, między innymi następujące:

- optyczne (oczy odbierają sygnały świetlne)
- dźwiękowe (uszy odbierają sygnały dźwiękowe)
- smakowe (język reaguje na substancje chemiczne)

W zależności od odebranych informacji wybieramy odpowiednie postępowanie, czyli odpowiednią reakcję. Na skutek otrzymanych informacji wykonujemy określone działania. Podejmowane działania mają ustaloną kolejność - odbywają się zgodnie z upływającym czasem.

Opisując nasze działania wygodnie jest podzielić wykonywany zestaw czynności na krótkie sekwencje (mówiąc inaczej rozpatrujemy nasze czynności, jako ciąg kroków).

Jako przykład rozpatrzmy zagadnienie poruszania się po mieście. Przypuśćmy, że w pewnym momencie chcemy przejść na drugą stronę ulicy. Jak w takiej sytuacji postępujemy? Zgodnie z odpowiednimi przepisami, przekraczamy ulicę w miejscu do tego przeznaczonym (przejście dla pieszych, odpowiednio oznakowane). Mogą wystąpić dwa przypadki:

a) na przejściu są specjalne światła sygnalizacyjne
b) na przejściu nie ma sygnalizacji świetlnej

Sposób przekraczania jezdni w obu przypadkach jest różny.

**Przypadek a.**

Gdy na przejściu zainstalowane są światła sygnalizacyjne, kolejność naszych działań jest następująca:

| Nr działania | Działanie |
|---|---|
| 1 | Zatrzymujemy się przy krawężniku |
| 2 | Obserwujemy lampy sygnalizacyjne |
| 3 | Jeżeli świeci lampa czerwona - stoimy |
| 4 | Jeżeli świeci lampa żółta - stoimy |
| 5 | Jeżeli świeci lampa zielona - przechodzimy |

**Przypadek b.**

Gdy na przejściu nie ma świateł sygnalizacyjnych, kolejność naszych działań jest następująca:

| Nr działania | Działanie |
|---|---|
| 1 | Zatrzymujemy się przy krawężniku |
| 2 | Patrzymy w lewą stronę |
| 3 | Jeżeli jedzie samochód - stoimy |
| 4 | Jeżeli nie jedzie samochód - patrzymy w prawą stronę |
| 5 | Jeżeli nie jedzie samochód - przechodzimy |
| 6 | Jeżeli jedzie samochód - stoimy |
| 7 | Jeżeli samochód minie nas powtarzamy całą procedurę (sposób postępowania) od kroku nr 2 |

Opisany przykład ilustruje zagadnienie **rozwiązania zadania**. Opisany zestaw czynności jest rozwiązaniem naszego zadania. Rozwiązanie zadania powinno mieć uzasadnienie - **dowód poprawności.** Musimy udowodnić, że nasze rozwiązanie jest prawdziwe. Jest to często bardzo trudny problem.

Określając zadanie do rozwiązania musimy także udowodnić (lub sprawdzić), czy nasze zadanie jest rozwiązywalne. Trudno jest będąc na pustyni, gdzie nie ma ulic, żądać od kogoś przejścia na drugą stronę.

Musimy także określić, w jakich warunkach, (przy jakich założeniach) zadanie jest rozwiązywalne. W opisie naszego rozwiązania milcząco zakładamy, że odróżniamy barwy. Metoda rozwiązania w przypadku a) jest kompletnie niezrozumiała dla daltonisty. W takiej sytuacji musimy zmienić rozwiązanie. Ważnym problemem jest także sprawdzenie, czy podane rozwiązanie gwarantuje otrzymanie wyniku w skończonym czasie jest to tzw. **problem stopu**. Łatwo możemy sobie wyobrazić sytuację (opisaną w przypadku b), że gdy chcemy przekroczyć ruchliwą ulicę w godzinach szczytu - postępując według podanego przepisu będziemy stać długo przy krawężniku. Zakończenie opisanych czynności może w pewnych przypadkach być niemożliwe. Podane rozwiązanie zadania musi być także **jednoznaczne**. Chodzi o to, że biorąc pod uwagę wybrany zestaw warunków (a czynników istotnych dla poprawnego rozwiązania zadania może być więcej) otrzymany wynik zawsze będzie taki sam. Podając rozwiązanie naszego zadania nie uwzględniliśmy np. faktu, że ulica może być dwupasmowa. Postępując zgodnie z naszym rozwiązanie przekroczymy tylko połowę ulicy!

W matematyce, fizyce czy chemii posługujemy się **wzorami matematycznymi**. Wzór matematyczny jest to skrótowy zapis związku, jaki istnieje między jakimiś wielkościami. Zwykle posługujemy się odpowiednimi symbolami matematycznymi. Wszystkie symbole występujące we wzorze muszą być dobrze zdefiniowane.

Wzorami posługujemy się, gdy chcemy rozwiązać zadanie. Niech nasze zadanie ma postać:

*Oblicz długość przeciwprostokątnej trójkąta prostokątnego mając dane długości przyprostokątnych a i b ; a = 1, b = 2 .*

Chcąc rozwiązać to zadanie korzystamy ze wzoru Pitagorasa. Metoda postępowania może być następująca:

| Nr działania | Działanie | Zapis działania |
|---|---|---|
| 1 | podnieś a do kwadratu | a*a |
| 2 | podnieś b do kwadratu | b*b |
| 3 | dodaj te kwadraty | a*a + b*b |
| 4 | oblicz pierwiastek kwadratowy | sqrt(a*a + b*b) |

Analiza tego prostego zadania pozwala nam na sformułowanie kilku równoważnych definicji algorytmu:

**Definicja A :**

*Algorytm, to taki zapis, który przedstawia wszystkie operacje, jakie należy wykonać (zachowując odpowiednią kolejność), aby rozwiązać zadanie nazywamy algorytmem.*

**Definicja B :**

*Algorytm jest to zbiór określonych reguł postępowania, które realizowane zgodnie z ustalonym porządkiem umożliwiają rozwiązanie określonego zadania*

**Definicja C :**

*Algorytm jest zbiorem instrukcji, które są odpowiednio zorganizowane i wystarczająco precyzyjne tak, że prowadzą do rozwiązania problemu.*

Zadania, które rozwiązujemy za pomocą komputera (przetwarzając informacje) możemy w ramach naszych wykładów podzielić na dwie grupy:

- zadania z dziedziny obliczeń numerycznych
- zadania z dziedziny przetwarzania danych

Metody numeryczne są klasą metod służących do rozwiązywania szeroko pojętych problemów matematycznych. Ta klasa metod jest niezwykła z tego powodu, że wykorzystywane są jedynie operacje arytmetyczne i logika. Oczywiście problemy matematyczne, które chcemy rozwiązać mogą być modelami matematycznymi dowolnego fizycznego zagadnienia. Zagadnienia numeryczne charakteryzują się stosunkowo małą ilością danych wejściowych, wymagają jednak poprawnej a najczęściej skomplikowanej i zaawansowanej analizy matematycznej, korzystają z potężnego aparatu metod numerycznych.

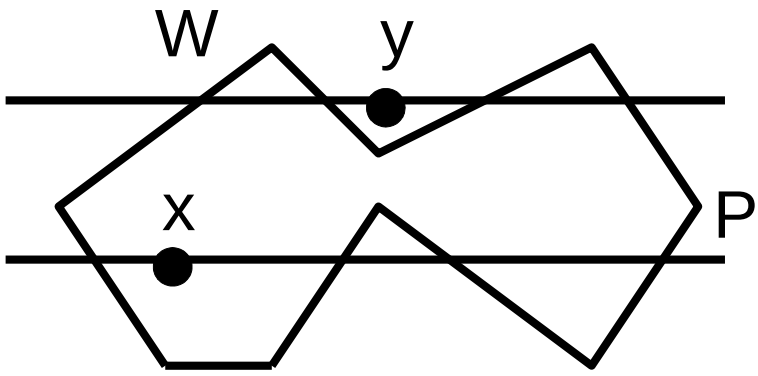
Zadania związane z przetwarzaniem danych charakteryzują się dużą ilością danych wejściowych i rozbudowanym sposobem prezentowania danych wyjściowych. W tego typu zadaniach analizujemy dokumenty wejściowe a następnie tworzona jest baza danych. **Baza danych** jest to uporządkowany zbiór danych. **System baz danych** jest to baza danych razem z oprogramowaniem, które umożliwia wykonywanie żądanych operacji na istniejących danych. Otrzymane dane wyjściowe prezentujemy w postaci odpowiednich zestawień wynikowych - najczęściej w postaci odpowiedniej grafiki lub tabel.

Algorytm możemy przedstawić na różne sposoby. Często algorytm przedstawia się, jako schemat blokowy. **Schemat blokowy algorytmu** jest to graficzne przedstawienie algorytmu. Elementarne czynności, jaki zawarte są w opracowanym algorytmie przedstawiane są za pomocą odpowiednich symboli graficznych. Istnieje polska norma definiująca te symbole (PN-75.E-01226).

Algorytm może być przedstawiony także w postaci pseudokodu. Pseudokod jest ogólnym i fundamentalnym sposobem zapisu algorytmu, bez korzystania z konkretnego języka programowania. Dzięki pseudokodowi otrzymujemy szczegółowy opis działania algorytmu, a ponieważ zapis jest wyrażony w języku naturalnym jest on zrozumiały dla człowieka, w dodatku uniezależniamy

się od konkretnego języka programowania. Algorytm wyrażony w postaci pseudokodu zazwyczaj stosuje się w początkowej fazie projektowania, gdy nie mamy jeszcze pewności, jaką zastosujemy technikę (np. czy korzystać z pętli **for** czy może z pętli **while**). Zaprezentujemy klasyczny przykład pseudokodu, który pochodzi z monografii F. Preparaty i M. Shamosa. Zadanie polega na określeniu czy dany punkt x na płaszczyźnie leży wewnątrz wielokąta W, czy na zewnątrz, wielokąt ma M kątów. Istnieje prosty algorytm rozwiązania tego zagadnienia. Przez punkt x prowadzimy prostą P. Jeżeli prosta P nie przecina wielokąta P, to punkt x leży na zewnątrz wielokąta. Jeżeli prosta P przecina W należy policzyć liczbę przecięć P z brzegiem w na lewo od punktu x (rys.1.1).
Algorytm zapisany w postaci pseudokodu może mieć postać:

```
begin N:= 0;
   for i:= 1 until M do
      if krawędź (i) nie jest pozioma then
         if(krawędź(i) przecina P na lewo od x w
            dowolnym punkcie innym niż początek) then
           N := N + 1;
         if (N jest nieparzysta) then
           x jest wewnątrz
         else
           x jest na zewnątrz
end
```




Rys. 1.1 Test zawierania się punktu w wielokącie W.

Widzimy, że prosta P przechodząca przez punkt x przecina wielokąt W tylko jeden raz (na lewo od x). Potwierdza się obserwacja, że punkt x leży wewnątrz W. Test dla punktu y wykazuje dwa przecięcia, punkt y nie leży wewnątrz wielokąta W. Przechodzenie prostej P przez któryś z wierzchołków wielokąta W wymaga odrębnej analizy. Analizując pokazany algorytm , można wykazać, że zostanie on wykonany w czasie O(M).

## 1.2. Informacje o językach programowania

Studenci informatyki lub innych pokrewnych kierunków, podejmują naukę programowania komputerów. Muszą zdecydować, w jakim języku programowania będą się specjalizować.

Programowanie jak sama nazwa wskazuje jest to tworzenie programów. Język programowania jest to specjalny język opisujący zadanie i sposób jego wykonania przez komputer. Każdy język programowania jest określany jego składnią i semantyką, są to ścisłe i precyzyjne zasady. Istnieje kilka podziałów języków programowania. Jednym z najprostszych podziałów jest podział na języki niskiego i wysokiego poziomu. Języki niskiego poziomu to te, które przypominają bardziej język maszyny (ciąg jedynek i zer), a języki wysokiego poziomu to te, które bardziej przypominają język używany przez człowieka (zazwyczaj angielski, bo to jest podstawowy język używany w informatyce).

Historycznie możemy wyróżnić 4 generacje języków (podział ten jest bardzo arbitralny):

1. Pierwsza generacja – języki te powstały w momencie zbudowania pierwszych komputerów (lata czterdzieste). W tym przypadku pisanie programu odbywa się bezpośrednio w języku maszyny – w trakcie programowania powstaje ciąg jedynek i zer.
2. Druga generacja – usprawnienie pisania programów składających się z zer i jedynek, poszczególne grupy cyfr zastępowane są zestawami liter. Powstają tzw. języki asemblerowe. Jest to generacja wspomnianych wcześniej języków niskiego poziomu
3. Trzecia generacja - jest to część języków wysokiego poziomu, uznawanych za nie-obiektowe. Są to normalne języki wysokiego poziomu (Fortran, Pascal, język C), ale nie są przystosowane do programowania obiektowego
4. Czwarta generacja jest to generacja języków obiektowych. Języki te z reguły są tylko rozwinięciem języków trzeciej generacji np. obiektowy C++ jest rozwinięciem języka C, podobnie Object Pascal jest rozwinięciem zwykłego Pascala, ale np. Java nie ma swojego nie-obiektowego odpowiednika. Pewne konstrukcje charakterystyczne dla języków obiektowych (np. szablony w języku C++) pozwalają na nowy typ programowania – **programowanie ogólne**. Programowanie ogólne (ang. *generic programming*), nazywane czasami programowaniem generycznym, polega na pisaniu kodu, który może działać z różnymi typami przekazywanymi do niego. Klasycznym przykładem programowania ogólnego jest pisanie programów korzystających z biblioteki STL (Standard Template Library), dołączanej do języka C++.

Języki programowania różnią się między sobą. Możemy je klasyfikować ze względu na sposób, w jaki są traktowane przez system komputerowy – mamy podział na języki interpretowane i języki kompilowane. Można je także klasyfikować także ze względu na obszar zastosowań.

***Język do przetwarzania danych*** jest przeznaczony do przekształcania zbiorów danych alfanumerycznych. Czołowym przedstawicielem tej grupy jest Cobol (ang. Common Business Oriented Language), zaprojektowany w 1959 roku. Cobol zaprojektowano tak, aby można było wydajnie pisać programy przetwarzające ogromne ilości danych. Główni użytkownicy to banki, instytucje rządowe i duże korporacje.

***Język do obliczeń naukowych*** stosowany jest do obliczeń numerycznych. Najbardziej znanym językiem tej grupy jest Fortran (ang. Formula Translator), który był jednym z pierwszych języków wysokiego poziomu, zaprojektowany i zrealizowany w połowie lat pięćdziesiątych. Fortran stosowany jest do obliczeń naukowych i inżynierskich.

***Język programowania systemowego*** jest używany do pisania systemów operacyjnych. Języki tego typu są często maszynowo-zależne, gdyż mają mechanizmy dostępu niskiego poziomu (np. bezpośrednie adresowanie rejestrów maszyny). Główni przedstawiciele tej grupy języków to język C, Bliss i BCPL.

***Język opisu zleceń*** pozwala na pisanie programów ułatwiających kontakt użytkownika z systemem operacyjnym. Reprezentantem języków tej grupy może być język VMS przeznaczony do obsługi systemu operacyjnego MVS stosowanego w komputerach IBM.

***Języki konwersacyjne*** pozwalają programiście na wykonywanie zmian w programie, który już jest wykonywany. Czołowi reprezentanci tej grupy to Lisp i APL. Lisp (ang. List Processing) został zaprojektowany w roku 1960, pod wpływem formalizmu matematycznego zwanego rachunkiem lambda. Lisp okazał się najbardziej udany w zastosowaniach obejmujących obliczenia symboliczne. Ponieważ programów w Lispie zazwyczaj się nie kompiluje, lecz się je interpretuje, jest on wygodny, jako język konwersacyjny.

***Język algorytmiczny*** charakteryzują się tym, że programista tworzy sekwencję niezbędnych instrukcji, które mają być wykonane w określonej kolejności. Prawie wszystkie praktycznie używane języki programowania są językami algorytmicznymi.

***Język obiektowy*** charakteryzuje się tym, że elementy danych są aktywne i mają one pewne własności związane zwykle z programem. Aktywne elementy danych odpowiadają na komunikaty, które powodują, że dane działają na sobie, w wyniku, czego powstają nowe elementy danych. Pierwszym takim językiem był Smalltalk. Idee zawarte w tym języku zostały przeniesione na język C++. Języki obiektowe wykorzystują metodologię programowania obiektowego (ang. object-oriented programming). Programowanie obiektowe wykorzystuje „obiekty" – są to elementy łączące **stan**, (czyli dane) i **zachowanie** (czyli procedury, zwane także metodami). Obiektowy program komputerowy jest zbiorem takich obiektów komunikujących się pomiędzy sobą w celu wykonania zadania.

Wybór języka programowania nie jest oczywisty, do wyboru mamy – język asemblerowy, język C, język C++, Phyton, PHP, Fortran, Cobol, itp.

Przeprowadzone w 1990 roku badania rynku niemieckiego pokazały, że 27 % programistów stosowało język C, 25% - stosowało Cobol, 10 % stosowało assembler, 7 % Fortran, 7 % - Pascal. W maju 2003 roku firma Borland przeprowadziła badania dotyczące wyboru języka programowania w Europie i Ameryce. Były to badania statystyczne, na ankietę odpowiedziało ponad 80 000 programistów. Wyniki tych badań prezentowane są w tabeli 1.

Tabela I. Popularność języków programowania (2003 rok).

| Język programowania | Procent użytkowników |
|---|---|
| Delphi | 28.85 |
| Java | 21.13 |
| C++ | 22.83 |
| C | 7.05 |
| Basic | 3.93 |
| Pascal | 3.17 |

Widzimy, że w roku 2003 dominowały trzy języki: Delphi (28.85 %), język C++ (22.83 %) i Java (21.13 %). Biorąc pod uwagę, że język C++ i Java wywodzą się z języka C, dominacja tej grupy języków programowania jest uderzająca.

W 2010 roku ranking popularności języków programowania prezentowany przez firmę TIOBE ( www.tiobe.com) miał następującą postać:

Tabela II. Popularność języków programowania (2010 rok).

| Język programowania | Procent użytkowników |
|---|---|
| Java | 17.51 |
| C | 17.28 |
| PHP | 9.91 |
| C++ | 9.61 |
| (Visual) Basic | 6.57 |
| C# | 4.26 |
| Python | 4.23 |
| Perl | 3.82 |
| Delphi | 2.68 |
| JavaScript | 2.65 |

Wysoka pozycja języka C odzwierciedla prawdziwą użyteczność tego języka.
Interesująco wygląda także lista najpopularniejszych języków programowania w ujęciu historycznym. W tabeli III pokazano, jak zmieniały się preferencje programistów w ciągu ostatnich dziesięciu lat. Niezmiennie wysoka pozycja języka C dobrze świadczy o jego sile.

Wydaje się rozsądne, aby dobrze wykształcony informatyk opanował następujące języki programowania (sądzimy, że jest to dobra rekomendacja na lata 2010 – 2015):

- język C/C++
- język Java
- PHP, Phyton

Należy pamiętać, że istnieje ogromna liczba różnych języków programowania. W ciągu dekady powstaje około 2000 nowych języków. Nie potrafimy odpowiedzieć na pytanie, dlaczego mamy tak wiele języków oprogramowania. W początkowej fazie programowania komputerów (lata 1950 -1970) w zasadzie chodziło o ułatwienie w pisaniu programów komputerowych, prezentowano różne podejścia do techniki programowania, powstawały takie języki jak LISP, Fortran, Cobol, Simula, Pascal, C.

Tabela III. Zmiany popularności języków programowania w latach 2000-2010.

| Język | Pozycja 2010 | Pozycja 2006 | Pozycja 2000 |
|---|---|---|---|
| Java | 1 | 1 | - |
| C | 2 | 2 | 1 |
| PHP | 3 | 4 | - |
| C++ | 4 | 3 | 9 |
| (Visual) Basic | 5 | 5 | 4 |
| C# | 6 | 7 | 22 |
| Python | 7 | 8 | - |
| Perl | 8 | 6 | 5 |
| Delphi | 9 | 9 | 7 |
| JavaScript | 10 | 10 | 13 |

Zgodnie z oszacowaniem B. Stroustrupa na świecie obecnie (2010 rok) żyje około 10 milionów zawodowych programistów, każdy z nich zazwyczaj zna kilka języków programowania, gdy zachodzi taka konieczność uczy się nowego języka. Uważa się, że 90 % wszystkich programów zostało napisanych w następujących językach: Ada, C, C++, C#, COBOL, Fortran, Java, PERL oraz PHP. Należy pamiętać, że każdy typ komputera rozumie tylko jeden język (komputer odczytuje instrukcje w formacie dwójkowym), zaliczany to grupy **języków niskiego poziomu**. Programista tworząc program komputerowy pisze tekst (jest to tzw. **program źródłowy**), ten tekst za pomocą innego programu komputerowego (interpretera lub kompilatora) jest przekształcany do postaci kodu dwójkowego. Można pisać programy bezpośrednio w postaci kodu dwójkowego, ale ponieważ format dwójkowy (zwany też binarnym) jest niewygodny, powstały języki bardziej zrozumiałe dla programisty, są to języki zaliczane do grupy **języków wysokiego poziomu** (takie jak Fortran, język C, C++, Java, itp.). Program źródłowy, pisany dla określonego komputera, może

mieć różną postać (używamy różnych języków programowania), ale po kompilacji lub interpretacji zostaną przekształcone do kodu zrozumiałego dla komputera. Różnice występujące pomiędzy językami można zilustrować prezentując kod w danym języku. Uruchomienie tych kodów, w wyniku spowoduje tą samą reakcję komputera. Pokażemy kody (programy), dzięki którym, na ekranie monitora pojawi się znany z każdego podręcznika programowania napis „Hello, World!".

1. Kod w języku Basic:

```
10 print "Hello, World"
```

2. Kod w języku Perl:

```
print "Hello, World;
```

3. Kod w języku Python:

```
print "Hello, World";
```

4. Kod w języku JavaScript:

```
<title>
   Hello. World w JavaScript
</title>
<script>
   alert(Hello, World")
</script>
```

5. Kod w języku Java:

```
class HelloWorld
{
   public static void main (String args[ ])
   {
      System.out.println("Hello, World");
   }
}
```

6. Kod w języku Fortran:

```
C  Program Hello
      write(*, 10)
      10 format('Hello, World')
END
```

7. Kod w języku Pascal:

```
Program Hello (Input, Output);
Begin
   Writeln ('Hello, World');
End.
```

8. Kod w języku C:

```
#include <stdio.h>
int main()
{
   printf("Hello, World");
   return 0;
}
```

9. Kod w języku C++:

```
#include <iostream>
int main()
{
   std :: cout << "Hello, World";
   return 0;
}
```

Pokazane kody są pisane za pomocą prostych interpreterów (1,2), bardziej rozbudowanych interpreterów (3,4,5) oraz pisane za pomocą kompilatorów (6,7,8,9). Różnice są wyraźnie widoczne, zadziwia różnorodność symboli i terminów, prawdziwa wieża Babel.

## 1.3. Zarys historii języka C/C++

Historia programowania (a tym samym języków programowania) jest dość długa i związana bardzo wyraźnie z postępem technicznym budowanych elektronicznych urządzeń liczących (komputerów). W miarę rozwoju architektury komputerów, metody pisania programów zmieniały się fundamentalnie. Przed rokiem 1954 prawie wszystkie programy były pisane w języku maszynowym. Można się zgodzić, że historia nowoczesnego programowania zaczyna się od zbudowania przez niemieckiego naukowca Konrada Zuse w 1935 roku pierwszego komputera i opracowania przez niego w 1946 roku pierwszego języka programowania o nazwie Plankalkul. Komputer Konrada Zuse, o nazwie Z-1, zbudowany był z prostych przekaźników i wykorzystywał dwójkowy system liczenia.

W 1949 roku opracowano język programowania o nazwie Short Code. W tym języku najpierw pisano ciąg instrukcji, które ręcznie były przekształcane na język maszynowy. Była to pierwsza realizacja techniki kompilacji.

W 1954 roku pracownik amerykańskiej firmy IBM, John Backus stworzył i kierował grupą naukowców, którzy opracowali język Fortran (nazwa wywodzi się ze słów FORmula TRANslator). Był to pierwszy komercyjny język wysokiego poziomu, ze względu na swoja uniwersalność odniósł błyskawiczny sukces.

Historia powstawania języka C a następnie C++ jest długa, rozwojem tych języków zajmowało się wielu wspaniałych fachowców. Język C powstał dzięki przejęciu podstawowych zasad z dwóch innych języków: BCPL i języka B. BCPL opracował w 1967 roku Martin Richards. Ken Thompson tworzył język B

w oparciu o BCPL. W 1970 roku na podstawie języka B opracowano system operacyjny UNIX w Bell Laboratories.

Twórcą języka C jest Dennis M. Ritchie, który także pracował w Bell Laboratories. W 1972 roku język C był implementowany na komputerze DEC PDP-11. Jedną z najważniejszych cech języka C jest to, że programy pisane w tym języku na konkretnym typie komputera, można bez większych kłopotów przenosić na inne typy komputerów.

Język C był intensywnie rozwijany w latach siedemdziesiątych. Za pierwszy standard przyjęto opis języka zamieszczony w dodatku pt. "C Reference Manual" podręcznika „The C Programming Language". Publikacja ukazała się w 1978 roku. Opublikowany podręcznik definiował standard języka C, reguły opisane w tym podręczniku nazywamy standardem K&R języka C. W 1983 roku Bell Laboratories wydało dokument pt. "The C Programming Language - Reference Manual", którego autorem był D. M. Ritchie. W 1988 Kernighan i Ritchie opublikowali drugie wydanie "The C Programming Language". Na podstawie tej publikacji, w 1989 roku Amerykański Narodowy Instytut Normalizacji ustalił standard zwany standardem ANSI języka C.

Za twórcę języka C++ (przyjmuje się, że jest to nadzbiór języka C) uważany jest Bjarne Stroustrup, który w latach osiemdziesiątych pracując w Bell Laboratories rozwijał ten język, tak, aby zrealizować programowanie obiektowe. Bjarne Stroustrup wspólnie z Margaret Ellis opublikował podręcznik „The Annotated C++ Reference Manual". W 1994 roku Amerykański Narodowy Instytut Normalizacji opublikował zarys standardu C++.

Nowym językiem, silnie rozwijanym od 1995 roku jest język programowania Java. Język Java opracowany w firmie Sun Microsystem, jest oparty na języku C i C++. Java zawiera biblioteki klas ze składnikami oprogramowania do tworzenia multimediów, sieci, wielowątkowości, grafiki, dostępu do baz danych i wiele innych.

Jak już to zasygnalizowano, język C jest podstawą języka C++ i Javy. Wobec tego należy doskonale znać ten język programowania.

Aby pisać programy komputerowe musimy dysponować kompilatorem wybranego języka programowania. Istnieje wiele doskonałych kompilatorów języków wysokiego poziomu. Po opanowaniu zasad programowania w języku C, opanowanie języka C++ (korzystając z tego samego kompilatora) nie będzie nastręczało zbyt dużo problemów.

Do nauki programowania potrzebne są podręczniki. Oczywiście istnieje wiele doskonałych podręczników do nauki programowania w języku C/C++. Można odwołać się do klasycznych podręczników, ale pamiętajmy, że podręcznik "Arkana C++" liczy 1082 strony a "Biblia Turbo C++" liczy 987 stron.

W 1997 roku ukazało się trzecie wydanie książki Bjarne'a Stroustrupa „Język programowania C++". Oczywiście język C++ jest nadal rozwijany, stąd uznano, że trzecie wydanie podręcznika B. Stroustrupa wyczerpująco ujmuje nowe elementy języka i *de facto* może być traktowane, jako standard.

## 1.4. Paradygmaty programowania

Może być wiele strategii programowania w języku C/C++. Generalnie należy przedstawić dwa paradygmaty programowania:

- Programowanie proceduralne
- Programowanie zorientowane obiektowo

Język C stosowany jest głównie w programowaniu proceduralnym i strukturalnym, język C++ stosuje się w programowaniu zorientowanym obiektowo. Język C++ jest nadzbiorem języka C. Oznacza to, że w każdym języku C++ zawarty jest język C.

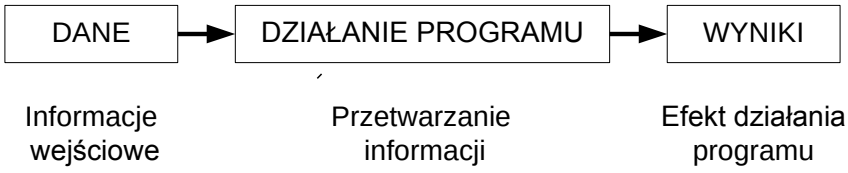
W programowaniu proceduralnym i strukturalnym program jest traktowany, jako seria procedur (funkcji) działających na danych. Dane są całkowicie odseparowane od procedur, programista musi pamiętać, które funkcje były wywołane i jakie dane zostały zmienione.

Przy realizacji dużych programów, dominuje programowanie obiektowe. Programowanie zorientowane obiektowo dostarcz technik zarządzania złożonymi elementami, umożliwia ponowne wykorzystanie komponentów i łączy w logiczną całość dane oraz manipulujące nimi funkcje. Zadaniem programowania zorientowanego obiektowo jest modelowanie „obiektów" a nie danych. Język C++ został stworzony dla programowania zorientowanego obiektowo. Zasadniczymi elementami stylu programowania obiektowego są: ukrywanie danych, dziedziczenie oraz polimorfizm.

Język C++ i Java są najbardziej popularnymi językami programowania obiektowego. Mimo, że wywodzą się z języka C, różnią się w wielu szczegółach.

## 1.5. Narzędzia programistyczne

Komputer za pomocą programu komputerowego przetwarza informacje. Działanie programu odbywa się według następującego schematu:




Rys.1.2 Przetwarzanie informacji

Nie wchodząc w szczegóły architektury (budowy) komputera, możemy powiedzieć, że najważniejszym elementem komputera jest mikroprocesor, który nadzoruje pracą wszystkich urządzeń. Oczywiście inne urządzenia, takie jak pamięć, czy urządzenia wejścia/wyjścia także są istotne dla działania komputera, ale o jego, jakości decyduje mikroprocesor.

Działanie mikroprocesora polega na możliwości wykonywania czterech operacji:

- Przeniesienie danych z jednego miejsca w pamięci do drugiego miejsca
- Zmienieniu danych w podanym miejscu pamięci
- Sprawdzeniu czy wskazane miejsce w pamięci zawiera żądane dane
- Zmienieniu kolejności wykonywanych czynności

Praca mikroprocesora oparta jest na sterowaniu sygnałami elektrycznymi: sygnał elektryczny może być włączony albo wyłączony. Mamy, zatem do czynienia z logiką dwuwartościową. Przyjęto oznaczać stan włączony, jako 1, stan wyłączony, jako 0. Możemy powiedzieć, że mikroprocesor „rozumie" wyłącznie polecenia dostarczone w postaci sekwencji zer i jedynek. Pisanie instrukcji dla procesora w postaci ciągów zer i jedynek nie jest zbyt atrakcyjne, – ale tak postępowano w początkowej fazie rozwoju komputerów. Następnym krokiem było opracowanie specjalnych programów, zwanych **językami asemblerowymi**. Programista za pomocą tzw. **kodów mnemotechnicznych** pisze program. **Asembler** następnie tłumaczy te kody na ciąg zer i jedynek. Asembler tłumaczy kody mnemotechniczne na sygnały elektryczne zrozumiałe dla mikroprocesora. Kody mnemotechniczne odnosiły się bezpośrednio do **instrukcji binarnych**. Pisanie programów w asemblerze było i jest zajęciem pracochłonnym i wymaga doskonałej znajomości procesora. Ostatnią ewolucją w metodach pisania instrukcji dla mikroprocesora było opracowanie tzw. **języków wysokiego poziomu**. Języki wysokiego poziomu, takie jak np. język C wykorzystują słowa zrozumiałe przez programistę, takie jak np. **do** ( wykonaj), czy **if** (jeżeli). Instrukcje napisane w języku wysokiego poziomu należy przetłumaczyć na ciąg zer i jedynek, aby były zrozumiałe dla mikroprocesora. Proces tłumaczenia instrukcji napisanych za pomocą języka wysokiego poziomu nosi nazwę **kompilacji**, a program, który to wykonuje nosi nazwę **kompilatora.** Formalnie proces tłumaczenia programu na ciąg bitów jest bardziej skomplikowany. Generalnie proces taki wykonywany jest w kilku krokach (rys.1.3).

Za pomocą **edytora tekstowego**, uwzględniając zestaw słów i gramatykę wybranego języka tworzymy program. Zapisujemy program w pliku tekstowym. Otrzymujemy tzw. **plik źródłowy** (kod źródłowy). Za pomocą programu zwanego **kompilatorem** dokonujemy wstępnego tłumaczenia a wynik tego tłumaczenia jest przechowywany w **pliku wynikowym** (kod obiektowy)

Za pomocą **programu łączącego** (konsolidator, linker) dołączane są tzw. **biblioteki** (programy niezbędne do działania naszego programu wyjściowego). Wynik łączenia przechowywany jest w **pliku wykonywalnym** (program wykonywalny). Ten program jest przekazany do mikroprocesora w postaci ciągu zer i jedynek a zatem może być wykonany.

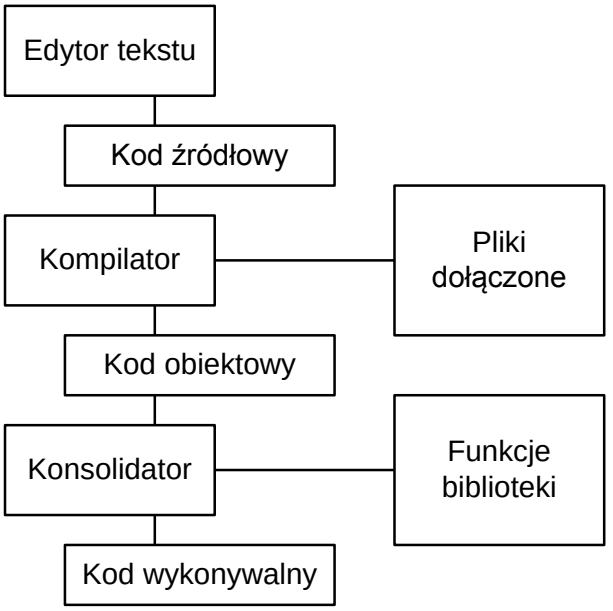
Opisane czynności pokazane są schematycznie na rys.1.3. Nie zawsze programista musi wykonywać oddzielnie wszystkie wymienione kroki.

Na wielu platformach kompilacja i łączenie mogą być wykonane w jednym kroku. Dla dowolnego kompilatora języka C/C++ dołączony jest opis techniczny mówiący jak dany kompilator wykorzystać. Napisany tekst programu za pomocą edytora musi być zapamiętany. Plik zapisujemy najczęściej na dysku twardym, najlepiej w odpowiednim katalogu. Jeżeli wykorzystujemy IDE to zapisanie tekstu źródłowego na dysku nie jest konieczne, kompilacja i uruchomienie programu odbywa się w środowisku IDE). Praktyka programowania sugeruje jednak, aby to zrobić. W początkowej fazie nauki języka C/C++ zawsze istnieje możliwość, że uruchomiony program zawiesi system – wtedy tekst źródłowy jest stracony. Plik tekstowy zapisujemy pod dowolną nazwą, rozszerzenie ma postać (najczęściej) albo .c albo .cpp. Następnym krokiem jest kompilacja. Kompilacja programu polega na przetłumaczeniu programu napisanego w języku zrozumiałym dla człowieka na język zrozumiały dla maszyny – tworzony jest inny plik – plik maszynowy (czasami nazywany **plikiem binarnym**). Ten krok byłby prosty, gdyby w wyniku kompilacji powstał plik wykonywalny. W wielu językach kompilowanych konieczny jest jeszcze jeden krok zwany łączeniem (konsolidacja, linkowanie).

Łączenie jest niezbędne z wielu powodów. Język C/C++ siłę swoją opiera na bibliotekach. Konieczne jest, zatem połączenie naszego programu z różnymi procedurami bibliotecznymi – np. funkcje które obsługują wejście i wyjście zawarte są w plikach nagłówkowych **<stdio.h>.** Drugim argumentem jest fakt, że program może składać się z kilku oddzielnych części – niektóre z nich mogą już być skompilowane, nie ma potrzeby, aby wykonywać powtórną kompilację. W procesie łączenia wszystkie niezbędne pliki są łączone w plik wykonywalny. W fazie kompilacji generowany jest **plik pośredni** (plik obiektowy). Dopiero konsolidator (program łączący) łączy wszystkie potrzebne pliki obiektowe w jeden – w rezultacie otrzymujemy plik wykonywalny.

W fazie kompilacji może wystąpić sygnalizacja błędu – kompilator nie potrafi zinterpretować jakichś np. instrukcji. Kompilacja zostaje przerwana – ukazują się **komunikaty o błędach** (mogą też wystąpić ostrzeżenia). Należy zanalizować komunikat i poprawić tekst źródłowy. Najczęściej mamy do czynienia z błędami składni lub błędy związane z interpunkcją. Po poprawieniu tekstu źródłowego proces kompilacji musi być powtórzony. Podobną sytuację możemy mieć w procesie łączenia – mogą wystąpić błędy. Należy przeanalizować komunikaty i usunąć usterki.

Jeżeli otrzymamy już plik wykonywalny, możemy przystąpić do najważniejszej fazy – testowania programu Ocenia się (arbitralnie), że 30 % czasu zajmuje kodowanie i 70% czasu testowanie. Testowanie programu ma za zadanie udowodnić, ze program działa poprawnie a otrzymywane wyniki są takie, jakich oczekiwaliśmy.




Rys.1.3 Schemat tworzenia pliku wykonywalnego

Ucząc się programowania trzeba się nauczyć składni (słów, gramatyki i interpunkcji) samego języka programowania. Ważne jest także przyswojenie zasad logiki programowania z użyciem komputera.

Do programowania w języku C potrzebny jest edytor, kompilator i konsolidator. Kompilator kupuje się wraz z konsolidatorem i zbiorem funkcji bibliotecznych. Istnieje wiele wersji kompilatorów języka C. Niektóre kompilatory języka C sprzedawane są razem z tzw. zintegrowanym środowiskiem do tworzenia aplikacji (ang. integrated development environment - IDE). Jest to bardzo wygodne narzędzie.

Wiele programów uruchamianych jest w systemie **Windows**. Pisanie przyjaznych dla użytkownika programów (takie programy charakteryzują się dużą ilością okienek i ikon) jest zagadnieniem dość skomplikowanym i uciążliwym technicznie. Aby usprawnić proces pisania programów dla środowiska Windows powstały systemy błyskawicznego projektowania aplikacji (ang. RAD – *Rapid Application Development*).

## 1.6. Struktura programu w języku C

Zaprezentujemy program napisany w języku C. W tym przykładzie chodzi nie o wyjaśnienie jak program został napisany czy co konkretnie robi, ale o zilustrowania ***ogólnej struktury*** programów pisanych w języku C. Programy w języku C mają bardzo podobną strukturę. Poniżej pokazano wydruk programu napisanego w języku C.

Listing 1.1. Ogólna struktura programu w języku C.

```
/* pm01 */
//struktura programu w jezyku C
#include <stdio.h>
#include <conio.h>              //clrscr(), getch()
#define PI 3.14159
#define KWADRAT(x) ((x)*(x))    //makro x*x
float pow_bocz(float, float);  //prototyp funkcji

int main()
{
   clrscr();                     //czysci ekran
   float promien,wysokosc;       //dane z klawiatury
   float pow_bocz_walca;
   printf("\npodaj promien : ");
   scanf("%f",&promien);
   printf("\npodaj wysokosc : ");
   scanf("%f",&wysokosc);
   pow_bocz_walca = pow_bocz(promien,wysokosc);
   printf("\npowierzchnia podstawy = %f",
           PI*KWADRAT(promien));
   printf("\npowierzchnia boczna walca = %f",
           pow_bocz_walca);
   getch();              // "przytrzymanie" ekranu
   return 0;
} //koniec funkcji main

//funkcja pow_bocz , oblicza powierzchnie boczna walca

float pow_bocz(float promien, float wysokosc)
{
   return(2*PI*promien*wysokosc);
}
```

Program napisany w języku C składa się z funkcjonalnych sekcji. Opiszemy bardziej szczegółowo te sekcje.

**Komentarze**

Na początku programu umieszczone zostały dwa ***komentarze***:

```
/* pm01 */
//struktura programu w języku C
```

Komentarz ignorowany jest przez kompilator (usuwany) w fazie kompilacji. Do zaznaczenia, że mamy do czynienia z komentarzem stosujemy specjalny sposób zapisu. W wielu kompilatorach mamy dwa sposoby zaznaczania komentarza. Symbol // oznacza, że komentarz rozpoczyna się od miejsca umieszczenia tego symbolu i rozciąga się do końca wiersza.
Symbol **/*** rozpoczyna komentarz, a symbol ***/** kończy komentarz.

Dzięki takiej symbolice możemy komentarze umieszczać w wielu wierszach. Należy pamiętać, że standard C99 dopuszcza użycie dwóch typów komentarzy:

```
/*……*/ oraz //
```

### Dyrektywy preprocesora

Następujący fragment programu:

```
#include <stdio.h>
#include <conio.h>                  //clrscr(), getch()
#define PI 3.14159
#define KWADRAT(x) ((x)*(x))        //makro x*x
```

zawiera ***dyrektywy preprocesora***. Instrukcje programu są instrukcjami dla mikroprocesora, dyrektywy preprocesora są instrukcjami dla kompilatora. Preprocesor jest w zasadzie wyspecjalizowanym automatycznym edytorem tekstowym. Wykorzystanie preprocesora jest wyjątkową właściwością języka C a także języka C++, rzadko spotykaną w innych kompilatorach. Dyrektywy preprocesora zawsze rozpoczynają się od znaku # (hasz). Istnieje wiele dyrektyw preprocesora. Tutaj zastosowano dwie – **include i define**. Dyrektywa **#include** powoduje wstawienie żądanego pliku do naszego pliku źródłowego. W pokazanym przykładzie żądamy, aby kompilator wstawił pliki nagłówkowe **stdio.h** i **conio.h**, pliki te zawierają funkcje biblioteczne. Te dyrektywy były nam potrzebne, aby zapewnić poprawne działanie funkcji **clrscr() i getch().**

Dyrektywa **#define** ma wiele zastosowań. W naszym programie spowodowaliśmy przypisanie nazwy **PI** stałej (= 3.14159). W kolejnej dyrektywie **#define** skorzystaliśmy z faktu, że może ona korzystać z parametrów. Jeżeli w programie preprocesor spotyka wyrażenie **KWADRAT(x)** zamienia je na instrukcję języka C : **x*x** (podnoszenie do kwadratu).

### Deklaracje zmiennych i funkcji

Kolejny wiersz programu:

```
float pow_bocz(float, float);   //prototyp funkcji
```

zawiera ***deklarację funkcji*** **pow_bocz().** Deklaracje informują, jakie zmienne i funkcje będą stosowane w programie, określają także typy danych .

### Funkcja main()

Wszystkie programy w języku C są podzielone na jednostki zwane funkcjami. Każdy program w C składa się z funkcji. Wobec tego musi istnieć jakaś wyróżniona funkcja, która musi być wykonywana pierwsza. Jest to jedyna funkcja, do której przekazywane jest sterowanie z systemu operacyjnego przy uruchamianiu programu. Nazwa **main()** jest zastrzeżona, inne funkcje możemy nazywać dowolnie. Pisząc **main(),** (nasz napis to literał i nawiasy okrągłe) specyfikujemy, że **main()** jest nazwą funkcji, a nie np. nazwą zmiennej. Słowo **int** oznacza, ze funkcja **main()** zwraca wartość całkowitą.

Następujący fragment programu:

```
int main()
{
   clrscr();                    //czysci ekran
   float promien,wysokosc;      //dane z klawiatury
   float pow_bocz_walca;
   printf("\npodaj promien : ");
   scanf("%f",&promien);
   printf("\npodaj wysokosc : ");
   scanf("%f",&wysokosc);
   pow_bocz_walca = pow_bocz(promien,wysokosc);
   printf("\npowierzchnia podstawy = %f",
                          PI*KWADRAT(promien));
   printf("\npowierzchnia boczna walca = %f",
                               pow_bocz_walca);
   getch();                 // "przytrzymanie" ekranu
   return 0;
} //koniec funkcji main
```

zawiera funkcję **main().** Funkcja **main()** zwraca wartość do systemu operacyjnego. Zanim wprowadzono standard ANSI C, kompilatory akceptowały napis:

```
main()
```

Pusty nawias jest interpretowany jako brak jakichkolwiek informacji o danych wymaganych przez **main().** Pominięcie słowa **int** najczęściej nie ma żadnych konsekwencji, ponieważ wiele kompilatorów języka C, zakłada, że funkcja zwraca wartość **int** (całkowitą). Zgodnie z ostatnimi standardami języka, zaleca się jednak używanie specyfikacji

```
int main()
```

Po nazwie funkcji znajdują się nawiasy klamrowe **{ }**. Wewnątrz tych nawiasów znajduje się blok kodu funkcji.

### Inne funkcje

W ostatnie części naszego programu mamy następujący fragment:

```
//funkcja pow_bocz , oblicza powierzchnie boczna walca
float pow_bocz(float promien, float wysokosc)
{
  return(2*PI*promien*wysokosc);
}
```

Pokazana jest tu inna funkcja o nazwie **pow_bocz().** Jest to definicja funkcji wywoływanej z poziomu funkcji **main().** Na początku programu umieszczona została deklaracja tej funkcji. Obecne standardy języka zalecają stosowanie deklaracji, zwanej czasami prototypem funkcji. Funkcja **main()** korzysta z tej funkcji.

Typowy program napisany w języku C składa się z następujących części:

- dyrektyw preprocesora
- deklaracji zmiennych i funkcji
- funkcji **main()**
- innych funkcji

**Siedem kroków programowania**

Język C jest językiem kompilowanym. W takim przypadku proces pisania programów w tym języku można podzielić schematycznie na siedem podstawowych kroków .

1. **Określenie celów programu**. Przede wszystkim należy wiedzieć co program ma zrobić. W tym kroku należy określić dane, jakich program będzie potrzebował, jakie zadania obliczeniowe mają być wykonane, jakie informacje mają być przekazane użytkownikowi.
2. **Projektowanie programu**. Wiedząc, jakie cele ma realizować program komputerowy należy zadecydować o sposobie jego realizacji. Określamy sylwetkę użytkownika, rodzaj interfejsu użytkownika, sposób organizacji programu. Należy ustalić sposób reprezentacji danych w programie. Należy też określić czas projektowania i realizacji programu.
3. **Pisanie kodu**. Gdy jest już gotowy projekt programu następuje faza implementacji. W tym przypadku implementacja oznacza pisanie kodu źródłowego. Pisanie kodu jest zależne od używanego środowiska programistycznego. Zazwyczaj pisząc kod źródłowy korzystamy z edytora tekstowego.
4. **Kompilacja**. Gotowy kod źródłowy musi być skompilowany. Szczegóły techniczne tego kroku zależą od używanego środowiska programistycznego. Wyspecjalizowany program zwany kompilatorem tłumaczy kod źródłowy na kod maszynowy. Różne komputery mają różne kody maszynowe. Kompilator języka C musi przetłumaczyć kod źródłowy w języku C na kod maszynowy konkretnego komputera. Kompilator nadzoruje także proces dołączania bibliotek. Ściślej mówiąc kompilator wywołuje automatycznie inny wyspecjalizowany program zwany programem łączącym (lub linkerem). Ostatecznie kompilator produkuje plik wykonywalny zawierający kod zrozumiały dla komputera. Ważną rolą kompilatora jest sprawdzanie poprawności napisanego programu. Wszystkie błędy są wychwytywane. W przypadku wystąpienia błędu proces kompilacji jest przerwany i wysyłany jest komunikat o błędzie.
5. **Uruchomienie programu**. Jeżeli proces kompilacji przebiegnie bez błędów powstaje kod wykonywalny, który można uruchomić. Aby uruchomić program w systemach Unix (Linux) wystarczy w tym celu napisać jego nazwę. W zintegrowanych środowiskach programistycznych (IDE) uruchomić możemy programy przy w ramach pakietu IDE. Najczęściej wystarczy kliknąć myszką odpowiednia opcję lub nacisnąć odpowiedni klawisz.

6. **Testowanie i usuwanie błędów**. To, że program daje się uruchomić nie oznacza, że działa poprawnie lub zgodnie z oczekiwaniami użytkownika. Należy wykonać serię testów, aby się upewnić, że program działa poprawnie. Testowanie programów jest skomplikowanym zagadnieniem, w zasadzie nie istnieją idealne rozwiązania i metody wykazania poprawności złożonego programu.
7. **Pielęgnowanie i modyfikacja programu**. W trakcie użytkowania programu mogą ujawnić się nie wykryte błędy. Z czasem może też pojawić się konieczność rozszerzenia programu o dodatkowe funkcje. Wszystkie te zabiegi będą łatwiej wykonane, gdy będzie istniała dobra dokumentacja programu.

Zalecana metoda postępowania jest ideałem. Wszyscy uznają, że należy kolejno wykonać siedem opisanych kroków, ale najczęściej początkujący programista zaczyna od kroku trzeciego, czyli od pisania kody źródłowego. W trakcie nauki programowania, gdy demonstrowane są proste i krótkie programy nie ma w tym nic złego. Przystępując do projektowania i implementacji złożonych projektów programistycznych należy jednak bezwzględnie przestrzegać siedmiu kroków poprawnego programowania.

We wczesnych latach 80-tych język C był dominującym językiem programowania w świecie minikomputerowych systemów uniksowych. Należy zwrócić uwagę, że w ostatnich latach wielu producentów oprogramowania zaczyna preferować język C++, programowanie obiektowe (w przeciwieństwie do strukturalnego) odgrywa coraz większą rolę. Język C++ jest niemal dokładnym nadzbiorem języka C, a to oznacza, że każdy poprawny program napisany w języku C jest także poprawnym programem w języku C++.

# ROZDZIAŁ 2

## KRÓTKI PRZEGLĄD JĘZYKA ANSI C

## 2.1. Wstęp

W tym rozdziale omówimy zasadnicze elementy języka C koncentrując się na następujących zagadnieniach:

- algorytm i program komputerowy
- wejście i wyjście
- typy i operatory
- selekcja
- iteracja
- funkcje
- tablice
- napisy
- wskaźniki
- pliki

Program w języku C zawiera **funkcje.** Każdy program musi posiadać przynajmniej jedną funkcję o nazwie **main()**, nazwa **main** jest zastrzeżona, inne funkcje mogą mieć dowolne nazwy. Wykonywanie programu zaczyna się od funkcji **main().** Język C posiada ***biblioteki***, w których zgromadzone są funkcje obsługujące wiele kluczowych zadań takich jak umieszczanie tekstu na ekranie monitora, odczytywanie znaków z klawiatury, itp. Warunkiem skorzystania z funkcji bibliotecznych jest umieszczenie w programie informacji o bibliotece. W tym celu należy do programu dołączyć ***plik nagłówkowy.***

Program może rozwiązać zadanie nie komunikując się z użytkownikiem, ale na początku, nasz pierwszy program powinien poinformować nas o wyniku swojego działania wysyłając odpowiedni komunikat na ekran monitora. Przygotujemy pierwszy program, który wyświetli linie tekstu na ekranie.

> *UWAGA.*
> W języku C nie ma numeracji linii tekstu. Do celów dydaktycznych, w wielu programach, umieszczamy taką numerację.

Listing 2.1. Wyświetlanie komunikatu na ekranie monitora.

```
01   /* C*/
02   /* Wyswietlenie napisu na ekranie*/
03     #include <stdio.h>
04
05   int main()
06   {
07     printf("Zaczynamy kurs programowania w C");
08     return 0;
09   }
```

Wykonanie tego programu spowoduje wyświetlenie napisu "**Zaczynamy kurs programowania w C**" na ekranie monitora. Program o nazwie pm2_1.cpp składa się z linijek tekstu. Do celów dydaktycznych każda linijka ma numer (w rzeczywistym programie, numeracja nie występuje).
W liniach 01 i 02 mamy tekst:

```
/* C */
/* Wyswietlenie napisu na ekranie*/
```

Napisy zaczynają się od kombinacji znaków **/*** i kończą się znakami ***/**. Jeżeli tekst otoczony jest takimi znakami, to mamy do czynienia z ***komentarzem***. Komentarz jest informacją dla programisty, nie podlega kompilacji, praktycznie jest usuwany z kodu.
W linii 03 mamy napis:

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

Znak # i słowo kluczowe (w tym przypadku "**include**") tworzą ***dyrektywę preprocesora***. Język C zbudowany jest z właściwego języka i bibliotek. Biblioteki języka C są dużymi zbiorami funkcji (np. takich jak użyta funkcja **printf()**). Funkcje biblioteczne umieszczone są w wielu ***plikach***. Dyrektywa w linii 03 mówi kompilatorowi żeby pobrał cały plik o nazwie **stdio.h** i umieścił w tym miejscu programu. Plik o nazwie **stdio.h** zawiera między innymi funkcję **printf().** Linia 04 jest pusta. Wprowadzamy ją, aby tekst źródłowy był czytelniejszy. Linia 05 zawiera tekst:

```
int main()
```

Funkcja **main()** jest funkcją, od której zaczyna się wykonywanie programu. W języku C funkcja musi mieć ***zadeklarowany typ***. W tym przypadku napis **int** oznacza typ całkowity (***integer***). Funkcja rozpoznawana jest po nawiasach okrągłych **( ).** Po nazwie funkcji mamy zapis ***treści funkcji*** (inaczej ***blok***). Treść funkcji zawarta jest pomiędzy nawiasami klamrowymi **{ },** które umieszczone są w liniach 06 i 09. ***Definicja funkcji* main()** ma postać:

```
int main()
{
   printf("Zaczynamy kurs programowania w C");
   return 0;
}
```

W treści funkcji mamy napisy:

```
printf("Zaczynamy kurs programowania w C");
return 0;
```

Występująca instrukcja **return,** w tym kontekście oznacza wyjście z programu do środowiska wywołującego funkcję **main().** Funkcja **printf()** umieszczona w linii 07 służy w tym programie do wyświetlenia na ekranie napisu:

```
Zaczynamy kurs programowania w C
```

Funkcja **printf()** jest bardzo rozbudowana, wykorzystaliśmy w naszym programie jej minimalne możliwości. Dane, które są przesyłane do funkcji tworzą ***argumenty funkcji***. W naszym przykładzie, funkcja **printf()** otrzymała tylko jeden argument. Jest to ***łańcuch znaków***, w żargonie programistów nazywany ***stringiem***. Łańcuch zaczyna i kończy się znakiem cudzysłowu. Przydatnymi argumentami funkcji **printf()** są ***kody ukośnika*** (zwane czasami ***znakami ucieczki***). Aby np. zacząć wyświetlanie napisu na ekranie od nowej linii, należy użyć znaku **\n**. Możemy zastąpić instrukcję w linii 07 następującym napisem:

```
printf("\nZaczynamy kurs programowania w C");
```

Jest wiele kodów ukośnika. Najczęściej stosowane to:

| | |
|---|---|
| \a | dzwonek |
| \n | nowa linia |
| \r | powrót karetki |
| \t | tabulator poziomy |
| \v | tabulator pionowy |

Linie 07 i 08 są ***instrukcjami***. Poznajemy to po znaku średnika na końcu linii. Każde działanie w C jest instrukcją i kończy się średnikiem.
W linii 08 mamy napis:

```
return 0;
```

W języku C funkcja **main()** powinna zwrócić wartość typu ***integer***. W tym przypadku funkcja zwraca zero.
Programy użytkowe najczęściej wymagają dialogu. W trakcie wykonywania programu na ekranie mogą pojawić się napisy żądające danych, program przetwarza te dane i wyświetla wyniki.

## 2.2. Operacje wejścia i wyjścia

Najczęściej dane są wprowadzane z klawiatury. Mówimy o operacjach **wejście - wyjście**. W języku C mamy do wyboru wiele funkcji obsługujących operacje wejście/wyjście. Najczęściej używane to **printf(), scanf(), getche(), gets(), puts().** Mogą one być umieszczone w różnych plikach nagłówkowych (np. **getche()** jest w pliku **conio.h**). Najczęstsze zastosowania tych funkcji to:

| | |
|---|---|
| `printf()` | wyświetla dane na ekranie |
| `puts()` | wyświetla łańcuch na ekranie |
| `scanf()` | pobiera dane z klawiatury |
| `getche()` | pobiera znak z klawiatury i daje echo |
| `gets()` | pobiera łańcuch z klawiatury |

Ponieważ mamy różne typy danych, wymagają one stosowania odpowiednich mechanizmów ich przetwarzania. W następnym programie zilustrujemy to zagadnienie, przy okazji wykorzystamy różne funkcje wejścia/wyjścia.

Listing 2.2. Stosowanie funkcji wejścia/wyjścia.

```
01   /*C */
02   //liczby calkowite i znaki
03   #include <stdio.h>
04   #include <conio.h>
05   int main()
06   {
07     char imie[20];
08     int liczba;
09     clrscr();
10     printf("\nnapisz swoje imie :");
11     gets (imie);
12     printf("podaj ulubina liczbe :");
13     scanf("%d",&liczba);
14     printf("\n%s , twoja liczba to: %d",
                  imie,liczba);
15     getche();
16     return 0;
17   }
```

Program najpierw pyta, jakie jest imię użytkownika a następnie pyta, jaka jest jego ulubiona liczba. Potrzebne dane wprowadzane są z klawiatury. Po wykonaniu programu mamy odpowiedni napis na ekranie monitora.
W linii 04 mamy dyrektywę:

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

Dyrektywa ta jest potrzebna między innymi ze względu na funkcję **getche().**
W liniach 07 i 08 mamy następujące napisy:

```
char imie[20];
int liczba;
```

Program potrzebuje odpowiednich danych, są to ***zmienne*** w programie. Zmienna musi mieć nazwę (u nas są to **imie** i **liczba**) oraz określony typ. W liniach 07 i 08 mamy ***deklaracje zmiennych***. Deklaracja **char** rezerwuje miejsce w pamięci operacyjnej dla zmiennej typu znakowego, deklaracja **int** rezerwuje miejsce dla zmiennej typu całkowitego. W linii 09 mamy funkcję **clrscr().** Ta funkcja "czyści" ekran. Po jej wywołaniu z ekranu usuwane są wszystkie znaki, kursor ustawiany jest w lewym górnym rogu ekranu. W linii 11 mamy napis:

```
gets (imie);
```

Funkcja **gets()** służy do wczytywania łańcucha. W języku C nie jest zdefiniowany typ łańcuchowy, język obsługuje pojedyncze znaki. Ta funkcja pobiera znaki pisane na klawiaturze aż do wciśnięcia klawisza ENTER. Praktycznie użytkownik ma wrażenie, że wczytywany jest łańcuch. W linii 13 występuje funkcja **scanf():**

```
scanf("%d",&liczba);
```

Funkcja formalnie służy do pobrania ciągów znaków (w naszym przypadku z klawiatury) i ich konwersji na wartość zmiennych języka C, zgodnie z podanym formatem. Symbol "**%d**" zwany ***formatem łańcucha***, informuje funkcję, że spodziewanym wejściem będzie liczba całkowita, dziesiętna. Wprowadzona dana będzie przechowywana w zmiennej o nazwie **liczba**. Operator **&** umieszczony przed nazwą przekazuje adres zmiennej **liczba**. Inaczej mówiąc, do funkcji **scanf()** przekazujemy wskaźnik do zmiennej **liczba**, ponieważ funkcja ta potrzebuje bezpośredniego dostępu do wartości przechowywanej w zmiennej **liczba.**
W linii 14 mamy instrukcję:

```
printf("\n%s , twoja liczba to: %d",imie,liczba);
```

Jeżeli użytkownik wprowadził dane: **Wacek** i **13,** wykonanie tej funkcji ma następujący skutek:

```
Wacek, twoja liczba to 13
```

Specyfikator łańcucha:

```
"\n%s , twoja liczba to: %d"
```

zawiera następujące elementy:

| | |
|---|---|
| \n | napis rozpocznie się od nowej linijki |
| %s | dana jest łańcuchem (specyfikator s) |
| twoja liczba to: | ten napis będzie umieszczony na ekranie |
| %d | dana jest typu całkowitego |

Specyfikator łańcucha oddzielony jest przecinkiem od listy argumentów, są to nasze zmienne **imie** i **liczba.**
Jeżeli chcemy obsłużyć liczbę rzeczywistą to stosujemy specyfikator **%f.** Dla wyświetlenia liczby dziesiętnej, dla której rezerwujemy osiem miejsc i chcemy mieć wyświetlone dwie liczby po przecinku, możemy zastosować specyfikator:

```
%8.2f
```

W linii 15 umieszczono funkcję **getche().** Ta funkcja (sztuczka programistyczna) służy do "przytrzymania" ekranu. W wielu środowiskach programistycznych, wynik ukaże się na ekranie i zniknie. Funkcja **getche()** czeka na znak wprowadzony z klawiatury. Naciśnięcie dowolnego klawisza kończy program. Tego typu zabieg (na różnych platformach może mieć inną postać) nazywamy „przytrzymaniem ekranu”.

## 2.3. Typy danych i operatory

Język C rozróżnia typy danych. Kompilator inaczej traktuje dane całkowite, inaczej dane rzeczywiste. Wykorzystywane przez program dane muszą mieć określony typ. Z punktu widzenia kompilatora istotny jest rozmiar danej (w bitach), dla programisty ważny jest zakres wartości.

Podstawowe ***typy danych*** w języku C to:

| | |
|---|---|
| typ całkowity | int |
| typ rzeczywisty | float, double |
| typ znakowy | char |

Należy także pamiętać, że język C rozróżnia duże litery i małe. W tabeli 2.1 podano najczęściej używane typy danych, ich rozmiary i zakresy wartości. Należy pamiętać, że te wielkości zależą od procesora i typu kompilatora. Aby mieć pewność, należy sprawdzić te dane w opisie technicznym procesora i kompilatora, można też skorzystać ze standardowych plików nagłówkowych **<limits.h>** oraz **<float.h>.** Oczywiście lista wszystkich dostępnych typów danych w konkretnym kompilatorze jest zazwyczaj znacznie większa, należy pamiętać o naturalnej skłonności producentów kompilatorów do tworzenia dialektów języka.

Tabela 2.1.

| Typ danych | Rozmiar (bajty) | Zakres wartości |
|---|---|---|
| char | 1 | -128 do 127 |
| unsigned char | 1 | 0 do 255 |
| int | 4 | -2147483648 do 2147483647 |
| unsigned int | 4 | 0 do 4294967295 |
| short int | 2 | -32768 do 32767 |
| long | 4 | tak samo jak **int** |
| float | 4 | $3.4*10^{-38}$ do $1.7*10^{38}$ |
| double | 8 | $1.7*10^{-308}$ do $3.4*10^{308}$ |

**Operatory przypisania**

Język C posiada operator przypisania (znak =) a także wiele operatorów przypisania, które są kombinacjami operacji arytmetycznych i przypisania. Ten dość nieczytelny zapis wprowadzono, aby programista mógł skracać wyrażenia i szybciej kodować. W tabeli 2.2 pokazana jest lista takich operatorów przypisania.

Tabela 2.2.

| Operator | opis | przykład |
|---|---|---|
| = | proste przypisanie | liczba = 6; |
| += | sumowanie i przypisanie | liczba += 10; |
| -= | odejmowanie i przypisanie | liczba -= 5; |
| *= | mnożenie i przypisanie | liczba *=10; |
| /= | dzielenie i przypisanie | liczba /= 100; |
| &= | złożenie AND i przypisanie | liczba1 &= liczba2; |
| \|= | złożenie OR i przypisanie | liczba1 \|= liczba2; |

**Operatory arytmetyczne**

Programy bardzo często wykonują operacje arytmetyczne. Język C nie posiada operatora potęgowego, zamiast tego w bibliotece standardowej znajduje się funkcja **pow(),** która wykonuje operacje potęgowania. Na przykład **pow(4.25, 2)** zwraca liczbę 4.25 podniesioną do potęgi 2).

Pisząc wyrażenia z operatorami arytmetycznymi, musimy pamiętać o kolejności wykonywania poszczególnych operacji arytmetycznych (np. operacja mnożenia ma pierwszeństwo przed operacja dodawania), w przypadkach wątpliwych najlepiej stosować nawiasy okrągłe.

W języku C lista operatorów arytmetycznych pokazana jest w tabeli 2.3.

Tabela 2.3.

| Operator | Opis | Przykład |
|---|---|---|
| + | dodawanie | 2 + 22 |
| - | odejmowanie | 100 - liczba |
| * | mnożenie | liczba * 3.33 |
| / | dzielenie | liczba1 / liczba2 |
| % | reszta z dzielenia całkowitego | liczba1 % 2 |
| -- | dekrementacja (zmniejszanie o 1) | i--, --j |
| ++ | inkrementacja (zwiększanie o 1) | k++, ++k |

Pokażemy program, który będzie zamieniał odległość mierzoną w milach i jardach na kilometry. Jedna mila to 1760 jardów. Jedna mila jest równa 1.609 km.

Listing 2.3. Operatory arytmetyczne.

```
01   /* C */
02   //operacje arytmetyczne
03   #include <stdio.h>
04   #include <conio.h>
05   int main()
06   {
07      float km;
08      int miles, yards;
09      miles = 26;
10      yards = 385;
11      km = (miles + yards/1760.0) * 1.609;
12      printf("\n maraton ma %12.4f km",km);
13      getche();
14      return 0;
15   }
```

W linii 07 oraz 08 umieszczono deklaracje zmiennych potrzebnych w programie. W deklaracji zmiennych specyfikujemy nazwę zmiennej oraz jej typ, może wystąpić lista zmiennych oddzielonych przecinkiem. Nazwy zmiennych muszą być unikalne ( nie mogą się powtarzać).

W linii 07 umieszczona jest deklaracja

```
float km;
```

Zmienna o nazwie **km** jest typu **float**, to znaczy jest liczbą rzeczywistą (w odróżnieniu od deklaracji w linii 08, gdzie deklarujemy zmienne całkowite). Zmienne rzeczywiste możemy deklarować także jako typu **double**, powoduje to zwiększenie precyzji obliczeń. W liniach 09 i 10 mamy ***instrukcję przypisania***:

```
miles = 26;
yards = 385;
```

Znak równości jest operatorem przypisania. Liczba 26 jest przypisana do zmiennej **miles**, a liczba 385 do zmiennej **yards.** Obie te zmienne są typu całkowitego.
W linii 11

```
km = (miles + yards/1760.0) * 1.609;
```

mamy także instrukcję przypisania. Z prawej strony operatora przypisania mamy ***wyrażenie***. Wartość tego wyrażenia jest najpierw obliczana a następnie przypisana zmiennej **km.** W tym wyrażeniu mamy operatory arytmetyczne takie jak dodawanie (+), dzielenie (/) i mnożenie (*). W zapisie mamy do czynienia z zagadnieniem kolejności wykonywania działań. Mówimy o ***priorytetach operatorów.*** Najwyższy priorytet mają nawiasy okrągłe, działania wewnątrz nawiasów są najpierw wykonywane. Wewnątrz nawiasów mamy dwa działania - dodawanie i mnożenie. Mnożenie ma większy priorytet niż dodawanie, zatem najpierw wykonane będzie dzielenie. Wynik dzielenia zostanie następny dodany do zmiennej **miles.** Ta suma będzie mnożona przez liczbę 1.609. Po wykonaniu programu na ekranie ukaże się napis:

```
maraton ma 42.1859 km
```

dzięki instrukcji napisanej w linii 12.
Generalnie należy unikać działań arytmetycznych na różnych typach zmiennych. W linii11 mamy dzielenie

```
 yards/1760.0
```

Zmienna yards jest typu **int**, stała 1760.0 jest typu **float**. Jeżeli zapiszemy stałą bez znaku dziesiętnego, (jako 1760) w wyniku dzielenia dostaniemy wartość 0. Aby poprawnie wykonać działanie w linii 12 kompilator dokonał niejawnej ***konwersji typu***. Przed dzieleniem zmienna **yards** (typ **int**) została promowana do typu **float.** Należy zachować dużą ostrożność w tego typu działaniach - jest to potencjalne źródło błędów. Można stosować jawne konwersje:

```
(float) yards/1760.0
```

Skorzystaliśmy z ***operatora zamiany typu*** (***rzutowanie***), chwilowo zmieniliśmy typ zmiennej.

**Operatory relacji i logiczne**
W programach często chcemy porównywać jakieś wielkości. W języku C mamy do dyspozycji zestaw operatorów **relacji** i **logicznych**. W odróżnieniu od operatorów arytmetycznych, ich argumenty oraz ich wyniki nie są liczbami, lecz wartościami logicznymi - **prawda** albo **fałsz** . W języku C wyrażenie takie jak

```
(8 == 2)
```

traktowane jest jak fałsz. Operatory relacji przedstawiono w tabeli 2.4.

Tabela 2.4. Operatory relacji.

| Operator | Opis |
|---|---|
| > | większy niż |
| >= | większy lub równy |
| < | mniejszy niż |
| <= | mniejszy lub równy |
| == | równy |
| != | nierówny |

Zestaw operatorów logicznych jest następujący:

| | |
|---|---|
| && | logiczne AND |
| \|\| | logiczne OR |
| ! | logiczne NOT |

## 2.4. Pętle w programach

W wielu przypadkach chcemy powtórzyć wykonywanie określonej części kodu. Do powtarzania wykonywania instrukcji lub bloku instrukcji służą ***pętle***. Zasadniczo mamy trzy możliwości:

pętla **for**
pętla **while**
pętla **do...while**

**Pętla for**
Jeżeli znamy ilość powtórzeń, najczęściej korzystamy z konstrukcji **for**. Pętla **for** jest bardzo elastyczna, możemy wewnątrz niej umieszczać wiele elementów i konstrukcji języka C.

Instrukcja **for** wymaga zazwyczaj trzech parametrów: zmiennej do zainicjowania licznika pętli, warunku trwania pętli, zmiany licznika. Licznik zazwyczaj ustawiany na 0, warunek jest sprawdzany za każdym razem, gdy ma być wykonana pętla. Gdy warunek jest fałszywy, pętla kończy działanie, a program wykonuje następną instrukcję. Pokażemy krótki program, który wykorzystuje pętlę **for**. W programie pokazanym na listingu 2.4 generujemy kolejne liczby całkowite. Pętla pokazana jest w liniach 08, 09, 10, i 11, instrukcja umieszczona jest w nawiasach klamrowych.

W linii 07 mamy inicjalizacje zmiennej **a**:

```
int a = 0;
```

W tym konkretnym przypadku nawiasy klamrowe nie są wymagane - można je opuścić. Nawiasy klamrowe stosuje się w przypadku, gdy w pętli mamy wykonać więcej niż jedną instrukcję.

```
for (int i=0; i<5; i++)
{
  printf("\n a =  %d",a++);
}
```

W linii 08:

```
for (int i=0; i<5; i++)
```

argumentami instrukcji **for** są: wartość początkowa licznika **i = 0,** warunek **i<5** oraz wyrażenie zwiększające zmienną **i** o 1, **i++** . W pętli zadeklarowano także zmienną **i** jako typu **int.**

Listing 2.4. Pętla **for.**

```
01   /* C */
02   //petla for
03   #include <stdio.h>
04   #include <conio.h>
05   int main()
06   {
07     int a = 0;
08     for (int i=0; i<5; i++)
09     {
10       printf("\n a =  %d",a++);
11     }
12     getche();
13     return 0;
14   }
```

Po wykonaniu tego programu na ekranie mamy następujący wynik:

```
a = 0
a = 1
a = 2
a = 3
a = 4
```

Jeżeli w funkcji **printf()** zmienimy zapis a++ na zapis ++a, otrzymamy:

```
a = 1
a = 2
a = 3
a = 4
a = 5
```

Pętla **for** w naszym przykładzie przypadku będzie powtórzona pięć razy. Pętla **for** jest bardzo elastyczna. W następnym przykładzie obliczymy wartość funkcji kwadratowej postaci:

$$y = 2x^2 - 3x - 4$$

Zmienna *x* przebiegać będzie wartości od -6 do +6 z krokiem 2. W wyniku działania naszego programu otrzymamy tablicę wartości funkcji dla żądanej wartości x.

Listing 2.5. Pętla **for**, generowanie tablicy wartości funkcji.

```
01   /* C */
02   //petla for
03   #include <stdio.h>
04   #include <conio.h>
05   int main()
06   {
07      int x;
08      float y;
09      for (x = -6; x <= 6; x = x + 2)
10      {
11         y = 2*x*x - 3*x - 4;
12         printf("%4d  \t \t  %8.1f \n",x,y);
13      }
14   getche();
15   return 0;
16   }
```

Po uruchomieniu programu otrzymamy wynik:

```
-6   86.0
-4   40.0
-2   10.0
 0   -4.0
 2   -2.0
 4   16.0
 6   50.0
```

W linii 09:

```
for (x = -6; x <= 6; x = x + 2)
```

licznikiem jest zmienna **x.** Po każdym cyklu pętli, wartość **x** zwiększa się o 2.
W linii 12 funkcja **printf()**

```
printf("%4d  \t \t  %8.1f \n",x,y);
```

otrzymała parametry formatujące **\t**, dzięki czemu wydruk stosuje tabulator.

**Pętla while**

Pętla działa podobnie jak pętla **for**, różni się brakiem licznika powtórzeń. Pętla **while** powtarzana jest dopóki spełniony jest warunek. Przed wykonaniem pętli (podobnie jak w przypadku pętli **for)** sprawdzany jest warunek. Jeżeli warunek nie jest spełniony, pętla się nie wykona. Oznacza to, że możliwy jest przypadek, że pętla **while** ( także **for**) w ogóle się nie wykona.

W programie z listingu 2.6 pokazano zastosowanie pętli **while.** Program wylicza kwadraty liczb 0, 1, 2, 3, 4 i 5.

Listing 2.6. Pętla **while.**

```
01   /* C */
02   //  petla while
03   #include <stdio.h>
04   #include <conio.h>
05   int main()
06   {
07      int x = 0;
09      while (x <= 5)
10      {
11         printf("kwadrat liczby %4d = %d \n",x,x*x);
12         x++;
13      }
14      getche();
15      return 0;
16   }
```

Po uruchomieniu programu na ekranie mamy następujący wynik:

```
kwadrat liczby 0  = 0
kwadrat liczby 1  = 1
kwadrat liczby 2  = 4
kwadrat liczby 3  = 9
kwadrat liczby 4  = 16
kwadrat liczby 5  = 25
```

W linii 07 zmiennej *x* nadano wartość 0. Pętla **while** realizowana jest w liniach 09, 10, 11, 12 i 13:

```
09   while (x <= 5)
10   {
11      printf("kwadrat liczby %4d = %d \n",x,x*x);
12      x++;
13   }
```

W linii 09 znajduje się słowo kluczowe **while** oraz warunek, który umieszczony jest pomiędzy nawiasami okrągłymi. Dopóki ten warunek jest prawdziwy, wykonywany jest blok instrukcji zawarty pomiędzy nawiasami klamrowymi - linie 11 i 12. W naszym przypadku, pętla będzie wykonywana tak długo jak **x**

będzie mniejsze lub równe 5. Ponieważ przed wykonaniem pętli sprawdzany jest warunek, musimy zmiennej **x** nadać wartość początkową. Wartość kwadratu oblicza wyrażenie **x*x** zawarte w funkcji **printf().** W linii 12 realizowana jest ***inkrementacja*** zmiennej **x** (zwiększanie jej wartości o 1 po każdym przebiegu pętli). Gdyby nie było tej instrukcji, pętla byłaby nieskończona.
W następnym przykładzie pokazujemy podobny przykład działania pętli **while.** Program ma wyliczyć pierwiastki liczb. Użytkownik podaje liczbę, od której program ma wystartować. W programie wykorzystano funkcję biblioteczną **sqrt()** do obliczania pierwiastka kwadratowego.

Listing 2.7. Pętla **while**, biblioteka **math.h.**

```
01   /* C */
02   //  petla while
03   #include <stdio.h>
04   #include <conio.h>
05   #include <math.h>
06   int main()
07   {
08     float x;
09     clrscr();
10     printf("\nliczba startowa: ");
11     scanf("%f",&x);
12     while (x >0)
13     {
14      printf("\n pierwiastek z liczby %8.2f = %8.2f",
                  x, sqrt(x));
15      x--;
16     }
17     printf("\nkoniec");
18     getche();
19     return 0;
20   }
```

W linii 05 umieszczono ***dyrektywę preprocesora***:

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

ponieważ w pliku **math.h** znajduje się funkcja **sqrt()** obliczająca pierwiastek kwadratowy. Pętla **while** realizowana jest w liniach 12, 13, 14, 15 i 16:

```
12   while (x >0)
13   {
14     printf("\n pierwiastek z liczby %8.2f = %8.2f",
                 x, sqrt(x));
15     x--;
16   }
```

Pętla będzie wykonywana dopóki warunek jest prawdziwy, to znaczy tak długo jak zmienna **x** jest większa od zera. W programie nie ma jawnego nadania

wartości początkowej zmiennej **x**. Wartość początkową tej zmiennej nadaje użytkownik za pomocą klawiatury:

```
10   printf("\nliczba startowa: ");
11   scanf("%f",&x);
```

Wyliczanie pierwiastka kwadratowego liczby realizuje funkcja **sqrt()** będącą parametrem funkcji **printf()**:

```
14  printf("\n pierwiastek z liczby %8.2f=%8.2f",x,sqrt(x));
```

Po uruchomieniu programu otrzymujemy żądanie wprowadzenia liczby:

```
        liczba startowa:
```

Po wprowadzeniu liczby np. 5 na ekranie mamy wynik:

```
pierwiastek z liczby    5.00 =    2.24
pierwiastek z liczby    4.00 =    2.00
pierwiastek z liczby    3.00 =    1.73
pierwiastek z liczby    2.00 =    1.41
pierwiastek z liczby    1.00 =    1.00
```

**Pętla do....while**

Działanie pętli **do ... while** nie różni się zbytnio od działania pętli **while** z wyjątkiem faktu, że sprawdzanie warunku następuje po wykonaniu pętli. W ten sposób mamy gwarancję, że pętla **do... while** wykona się, co najmniej jeden raz. W przykładzie z listingu 2.8 obliczamy pierwiastek kwadratowy liczby. Korzystamy z konstrukcji **do...while**. Użytkownik decyduje czy kontynuować obliczenia zmieniając wartość wyrażenia w warunku pętli.

W linii 09 zadeklarowano zmienną znakową **znak** i nadano jej wartość początkową. Należy zwrócić uwagę na sposób zapisu zmiennej typu **char.**

```
09   char znak = 't';
```

Pętla **do... whlie** realizowana jest w liniach 12, 13, 14, 15, 16, 17, 18 i 19 :

```
12   do
13   {
14    printf("\nliczba : ");
15    scanf("%f",&x);
16    printf("\n pierwiastek z liczby %8.2f =
                  %8.2f",x, sqrt(x));
17    printf("\n tak=t, nie=n, dalej liczyc : ");
18                      znak = getche();
19   } while (znak == 't');
```

W linii 12 umieszczone jest słowo kluczowe **do**.

Pętla **do...while** realizuje instrukcje zawarte pomiędzy nawiasami klamrowymi (linie 14,15,16,17 i 18). Po klamrowym nawiasie zamykającym umieszczone jest słowo kluczowe **while** z wyrażeniem warunkowym umieszczonym w nawiasach okrągłych.

Listing 2.8. Pętla **do…while.**

```
01  /* C */
02  //  petla  do ..while
03  #include <stdio.h>
04  #include <conio.h>
05  #include <math.h>
06  int main()
07  {
08    float x;
09    char znak = 't';
10    clrscr();
11    printf("\nwprowadzenie liczby ujemnej daje
              blad\n");
12    do
13    {
14      printf("\nliczba: ");
15      scanf("%f",&x);
16      printf("\n pierwiastek z liczby %8.2f =
                  %8.2f",x, sqrt(x));
17      printf("\n tak=t, nie=n, dalej liczyc: ");
18      znak = getche();
19    } while (znak == 't');
20    printf("\nkoniec");
21    getche();
22    return 0;
23  }
```

Wartość wyrażenia warunkowego decyduje o tym czy pętla **do…while** będzie wykonywana (linia 19):

```
(znak == 't');
```

Pętla będzie wykonywana dopóki to wyrażenie jest prawdziwe. W linii 18:

```
znak = getche();
```

znajduje się funkcja **getch(),** która pobiera znak wprowadzony z klawiatury. Zmiennej **znak** nadana jest odpowiednia wartość. Jeżeli wprowadzimy inny znak niż **'t',** warunek nie będzie spełniony. W wyrażeniu warunkowym należy zwrócić uwagę na operator ==. Częstym błędem jest zapominanie, że w tym miejscu powinien być podwójny znak równości. Po wykonaniu pierwszego przebiegu pętli, użytkownik pytany jest czy dalej ma kontynuować obliczenia. Jeżeli chce dalej liczyć, wprowadza znak **'t'** z klawiatury, jeżeli chce zakończyć - wprowadza dowolny inny znak, aczkolwiek program sugeruje wprowadzenie

znaku '**n**'. Po uruchomieniu programu na ekranie ukazuje się komunikat:

```
wprowadzenie liczby ujemnej daje blad
liczba:
```

Po wpisaniu np. liczby 25, na ekranie otrzymujemy wynik:

```
pierwiastek z liczby    25.00     = 5.00
tak = t, nie = n, dalej liczyć:
```

Jeżeli użytkownik wprowadzi z klawiatury znak "**t**" to wtedy program żąda wprowadzenia liczby:

```
liczba:
```

Po wprowadzeniu np. liczby 16 mamy wynik:

```
pierwiastek z liczby    16.00     = 4.00
tak = t, nie = n, dalej liczyć :
```

Jeżeli wprowadzimy znak '**n**', program zakończy działanie wypisując komunikat "**koniec**".

## 2.5. Instrukcje wyboru (selekcja)

Często wykonanie jakiejś części programu zależy od spełnienia odpowiednich warunków. Podczas rozwiązywania np. równania kwadratowego, sposób wyliczania jego pierwiastków zależy od wartości wyróżnika kwadratowego. Inaczej liczy się pierwiastki równania, gdy wyróżnik jest równy zero, a inaczej, gdy jest większy od zera. Język C dostarcza dwie podstawowe konstrukcje do obsługi decyzji - instrukcję **if** oraz instrukcję **switch.**

**Instrukcja if**

Instrukcja **if** należy do grupy instrukcji sterujących. Wartość wyrażenia warunkowego decyduje, która grupa instrukcji będzie wykonana. Jeżeli wartość wyrażenia jest prawdziwa, wykonywana jest pierwsza instrukcja (lub blok instrukcji), jeżeli wartość wyrażenia jest fałszywa, pierwsza instrukcja jest pomijana a wykonuje się druga. Program z listingu 2.9 pokazuje konstrukcję **if.** Program podaje wartość bezwzględną liczby podanej przez użytkownika. Instrukcja **if** umieszczona jest w linii 10 :

```
if (x < 0)     x = -x;
```

W nawiasach okrągłych znajduje się wyrażenie warunkowe. Jeżeli podana liczba jest większa od zera, instrukcja przypisania: x = -x nie jest wykonywana, sterowanie przekazane jest do następnej instrukcji:

```
printf("\nwartosc bezwzgledna = %d",x);
```

Listing 2.9. Konstrukcja **if.**

```
01   /* C */
02   //  instrukcja if
03   #include <stdio.h>
04   #include <conio.h>
05   int main()
06   {
07     int x;
08     printf("\npodaj liczbe : ");
09     scanf("%d",&x);
10     if (x < 0)    x = -x;
11     printf("\nwartosc bezwzgledna = %d",x);
12     getche();
13     return 0;
14   }
```

Jeżeli liczba jest ujemna, warunek jest prawdziwy i wykonywana jest instrukcja:

```
x = -x;
```

Formalnie konstrukcja **if** jest bardziej rozbudowana i ma postać:

```
if (wyrażenie)
  {blok instrukcji}
else
  {blok instrukcji}
```

W przykładzie z listingu 2.10 pokazano użycie tej konstrukcji.

Listing 2.10 Konstrukcja **if…else.**

```
01   /* C */
02   //instrukcja if
03   #include <stdio.h>
04   #include <conio.h>
05   #include <math.h>
06   int main()
07   {
08     int a, b, c, kw;
09     printf("\npodaj a, b i c : ");
10     scanf("%d %d %d",&a, &b, &c);
11     kw = a*a + b*b;
12     if (kw == c*c)
13       printf("\n trojkat prostokatny");
14     else
15       printf("\n to nie jest trojkat prostokatny");
16     getche();
17     return 0;
18   }
```

Program realizuje sprawdzanie czy trójkąt o bokach a, b i c jest trójkątem prostokątnym, korzystając z twierdzenia Pitagorasa. W linii 11 zmiennej **kw** nadana jest wartość wyrażenia $a^2 + b^2$. Konstrukcja **if..else** ma postać:

```
12 if (kw == c*c)
13   printf("\n trojkat prostokatny");
14  else
15   printf("\n to nie jest tojkat prostokatny");
```

W linii 12 sprawdzany jest warunek:

```
(kw == c*c)
```

Jeżeli ten warunek jest prawdziwy, wykonuje się instrukcja umieszczona w linii 13, jeżeli warunek jest fałszywy to wykonuje się instrukcja pokazana w linii 15.

**Instrukcja switch**

Instrukcje **if** mogą być ***zagnieżdżane*** (wewnątrz jednej instrukcji **if** umieszczona jest inna instrukcja **if**). Instrukcja **switch** daje możliwość obsługi decyzji wielowariantowych w dość prosty sposób.

Listing 2.11. Konstrukcja **switch.**

```
01   /* C */
02   //instrukcja switch
03   #include <stdio.h>
04   #include <conio.h>
05   int main()
06   {
07     int opcja;
08     printf("\nopcje:1-zapis,2-edycja,3-wykonanie");
09     printf("\npodaj opcje: ");
10     scanf("%d",&opcja);
11     switch (opcja)
12     {
13       case 1:
14        {printf("\n zapis pliku");
15         break;  }
16       case 2:
17        {printf("\n edycja pliku");
18         break;  }
19       case 3:
20        {printf("\n wykonanie pliku");
21         break;  }
22       default : printf("\n inna operacja");
23     }
24     getche();
25     return 0;
26   }
```

Pokazany przykład realizuje podjęcie odpowiedniej akcji (skierowania do bloku instrukcji) na podstawie wyboru jednej z trzech możliwości:

opcja nr 1 - zapis pliku
opcja nr 2 - edycja pliku
opcja nr 3 - wykonanie pliku

Użytkownik podaje numer opcji, program przekazuje sterowanie do odpowiedniego miejsca programu.
Konstrukcja **switch** :

```
switch (opcja)
{
   case 1:
   ........
}
```

wykorzystuje słowa kluczowe "**switch**" oraz "**case**", całość zamknięta jest w nawiasach klamrowych.
Konstrukcja **switch** umieszczona jest w liniach 11 - 23 :

```
11   switch (opcja)
12   {
13     case 1:
14       { printf("\n zapis pliku");
15         break;  }
16     case 2:
17       { printf("\n edycja pliku");
18         break;  }
19     case 3:
20       { printf("\n wykonanie pliku");
21         break;  }
22     default : printf("\n inna operacja");
23   }
```

Wyrażenie stojące za słowem kluczowym **switch** (w naszym przypadku jest to: **opcja**) określa możliwe wyjścia. Jeżeli np. **opcja** ma wartość 2 to sterowanie przekazane będzie do linii 16 gdzie znajduje się słowo kluczowe **case** z wyrażeniem stałym 2:

```
16   case 2:
17   {  printf("\n edycja pliku");
18     break;  }
```

Program wykona instrukcje zawarte pomiędzy nawiasami klamrowymi. Umieszczone w tym bloku słowo kluczowe "**break**" informuje program, że należy w tym miejscu opuścić instrukcję **switch** i przekazać sterowanie do pierwszej instrukcji następującej po zamykającym nawiasie klamrowym (w naszym przypadku jest to linia 24).

## 2.6. Tablice

Tablica jest jednorodną strukturą danych. Zawiera elementy tego samego typu, są one opatrzone tzw. ***indeksami.*** Należy zwrócić uwagę, że numerowanie elementów tablicy w języku C zaczyna się od zera. Pierwszy element tablicy ma indeks 0, co często prowadzi do nieporozumień i jest przyczyną błędów. Następująca deklaracja tablicy o nazwie **tab**:

```
int tab[4];
```

rezerwuje pamięć dla czterech zmiennych typu **int** do których możemy się odwołać jako **tab[0], tab[1], tab[2] i tab[3].**
Posługiwanie się tablicami zilustrujemy prostym przykładem. Przypuśćmy, że chcemy śledzić postępy uczniów. Zakładamy, że mamy pięciu uczniów, każdy z nich otrzymał jakąś ocenę. Zadaniem naszego programu jest zarejestrowanie stopni i znalezienie ucznia z najwyższą oceną. Program wczytuje stopnie kolejnych pięciu uczniów, szuka ucznia z najwyższą oceną i drukuje wynik. W programie są dwa ograniczenia - możemy wprowadzić dane tylko dla pięciu uczniów, jeżeli dwóch i więcej uczniów ma tą samą maksymalna ocenę, wyświetlany jest wynik tylko jednego ucznia.

Listing 2.12. Tablice jednowymiarowe.

```
01  /* C */
02  //  tablice
03  #include <stdio.h>
04  #include <conio.h>
05  int main()
06  {
07    int st[5];
08    clrscr();
09    for (int i=0; i<5; i++)
10    { printf("Nr %d podaj stopien : ",i);
11      scanf("%d",&st[i]);
12    }
13    int j=0;
14    int max = st[0];
15    for (i=0; i<5; i++)
16    if (st[i] > max) j = i;
17    printf("\nNajlepszy jest uczen nr %d i jest
                 to %d", j,st[j]);
18    getche();
19    return 0;
20  }
```

W linii 07:

```
int st[5];
```

zadeklarowano tablice o nazwie **st** która ma pięć elementów.

W liniach 09, 10, 11 i 12 widzimy pętle **for** służąca do wprowadzania danych z klawiatury. Za obsługę klawiatury odpowiada funkcja **scanf():**

```
11   scanf("%d",&st[i]);
```

W liniach 13 i 14 :

```
13   int j=0;
14   int max = st[0];
```

widzimy deklarację zmiennych **j** oraz **max** z inicjalizacją (nadaniem wartości). W liniach 15 i 16 :

```
15   for (i=0; i<5; i++)
16      if (st[i] > max) j = i;
```

znajduje się pętla **for** za pomocą, której poszukiwany jest element tablicy **st** o największej wartości. Wyrażenie warunkowe:

```
if (st[i] > max) j = i;
```

porównuje aktualny element tablicy **st** z wartością **max**. Zmienna **max** na początku ma wartość pierwszego elementu tablicy, **st[0].** Jeżeli kolejny element tablicy ma wartość większą, zapamiętywany jest indeks tego elementu. W ten sposób szukamy największego elementu tablicy. Funkcja **printf()** w linii 17:

```
printf("\nNajlepszy jest uczen nr%d i jest to %d",j,st[j]);
```

drukuje numer ucznia z najwyższą oceną i jego stopień.
Należy zauważyć, że pętla **for** może zaczynać się od i = 1, ponieważ porównanie elementu zerowego nie jest potrzebne. Jest wyraźna wada w programie – nie obsługujemy przypadku, gdy kilku uczniów ma ten sam stopień.
W języku C nie ma odrębnej obsługi **łańcuchów znakowych**. Programista może korzystać z bibliotek lub sam programować obsługę łańcucha. Łańcuch jest traktowany, jako tablica znaków. W programie pokazano sposób obsługi łańcucha znaków. Po uruchomieniu programu z listingu 2.13, na ekranie monitora ukazuje się napis:

```
Witaj, jak sie nazywasz?
```

Użytkownik wprowadza swoje nazwisko z klawiatury. Po wpisaniu np. Kowalski, na ekranie ukaże się komunikat:

```
pracujemy razem Kowalski
```

Program jest na pierwszy rzut oka skomplikowany - związane jest to ze specjalnym traktowaniem obsługi łańcuchów. Znaki przechowywane są w tablicy, na początku należy określić jej rozmiar. Możemy to zrobić bezpośrednio w deklaracji tablicy, możemy także wykorzystać dyrektywę preprocesora. W linii 05:

```
#define LINIA 100
```

stała **LINIA** ma wartość 100.

Listing 2.13. Tablice znaków.

```
01   /* C */
02   //  tablice znakow
03   #include <stdio.h>
04   #include <conio.h>
05   #define LINIA 100
06   int main()
07   {
08      char c, wier[LINIA];
09      clrscr();
10      printf("\n Witaj, jak sie nazywasz? ");
11      for (int i = 0; (c = getchar()) != '\n'; ++i)
12        wier[i] = c;
13      wier[i] = '\0';
14      printf("\n pracujemy razem ");
15      for (i = 0; wier[i] != '\0'; ++i)
16        putchar(wier[i]);
17      getche();
18      return 0;
19   }
```

Zakładamy, że użytkownik nie wprowadzi więcej niż 100 znaków. **LINIA** jest jednocześnie rozmiarem tablicy. Jest to bardzo wygodne, gdy użytkownik zechce wprowadzić np. 150 znaków, wtedy musi zmienić tylko tą wartość.
Znaki z klawiatury wprowadzane są pojedynczo, zatem musimy zadeklarować zmienną typu **char**, znaki te tworzą łańcuch przechowywany w tablicy, potrzebna jest nam zmienna tablicowa. Deklarację tych zmiennych widzimy w linii 08:

```
char c, wier[LINIA];
```

Tablica o nazwie **wier** ma rozmiar **LINIA** (w naszym przypadku jest to 100). Wprowadzanie tekstu obsługuje pętla **for**:

```
11   for (int i = 0; (c = getchar()) != '\n'; ++i)
12     wier[i] = c;
```

Wyrażenie warunkowe pętli ma postać:

```
(c = getchar()) != '\n';
```

Funkcja **getchar()** pobiera jeden znak z klawiatury i przypisuje go zmiennej c:

```
c = getchar()
```

Wykonywany jest test

```
c  !=  '\n'
```

w przypadku prawdy, elementowi tablicy o indeksie **i** przypisana jest wartość **c**. Następnie indeks **i** zwiększany jest o 1 i następuje kolejne wczytanie znaku. Zgodnie z przyjętą konwencją, w języku C wszystkie łańcuchy kończą się

znakiem **null ( \0 ).** W linii 13mamy:

```
wier[i] = '\0';
```

Ostatnim elementem łańcucha jest znak **null** (nie musi być to ostatni element tablicy, możemy wczytać tylko np. 10 znaków). Jeżeli każdy element tablicy wyobrazimy sobie, jako kratkę, to nasz łańcuch możemy przedstawić następująco:

| | J | a | n | | K | o | w | a | l | s | k | i | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Indeks | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Następna pętla **for**:

```
15 for (i = 0; wier[i] != '\0'; ++i)
16   putchar(wier[i]);
```

powoduje umieszczenie nazwiska (elementów tablicy **wier[])** na ekranie. Warunek w pętli sprawdza czy nie został napotkany znak końca łańcuch ( **\0** ). Umieszczanie znaków na ekranie obsługuje funkcja **putchar().**

## 2.7. Funkcje

Popularnie mówi się, że istotą programowania w C są **funkcje**. Funkcje są oddzielnymi ***blokami instrukcji***. Każdy program w C musi posiadać przynajmniej jedną funkcję - jest to funkcja **main().** Ta wyróżniona funkcja wywołuje inne potrzebne funkcje. Posługiwanie się funkcjami w języku C jest dość proste.
Chcemy opracować program, który będzie wykonywał następujące zadania:

1. Wyświetli informacje o programie
2. Wczyta z klawiatury dane liczbowe
3. Znajdzie wartość maksymalną
4. Pokaże na ekranie wynik

Program z listingu 2.14 pokazuje wykorzystanie funkcji w programie napisanym w języku C.
W liniach:

```
06   void info();       //informacja o programie
07   void dane();       // wprowadzanie danych
08   int licz();        // szuka najwieksza liczbe
09   void wyniki();     // podaje wynik
```

znajdują się tzw. ***prototypy funkcji.*** Każda funkcja powinna mieć określony typ, związany z typem zwracanej wartości. Słowo kluczowe **void** informuje, że funkcja nie zwraca żadnej wartości.

Listing 2.14. zastosowanie funkcji.

```
01  /* C */
02  //  funkcje przyklad 01
03  #include <stdio.h>
04  #include <conio.h>
05
06  void info();      //informacja o programie
07  void dane();      // wprowadzanie danych
08  int licz();       // szuka najwieksza liczbe
09  void wyniki();    // podaje wynik
10
11  int n, x[100], max;
12
13  int main()
14  { clrscr();
15    info();
16    dane();
17    licz();
18    wyniki();
19    return 0;
20  }
21  void info()
22  { printf("\n%s\n%s",
23    "Uwaga: program wczytuje n liczb calkowitych",
24    " znajduje wartosc maksymalna");
25  }
26  void dane()
27  { printf("\n ile liczb n = ");
28    scanf("%d",&n);
29    printf("\n wprowadz liczby ");
30    for (int j=0; j<n; j++)
31    scanf("%d",&x[j]);
32  }
33  int licz()
34  { max = x[0];
35    for (int i = 1; i<n; i++)
36    if (x[i]>max) max = x[i];
37    return max;
38  }
39  void wyniki()
40  { printf("\n maksymalna wartosc to %d",max);
41    getche();
42  }
```

W linii 11:

```
int n, x[100], max;
```

zadeklarowano zmienne używane przez funkcje **main()** jak i pozostałe funkcje.

Umieszczenie ich przed wywołaniem funkcji **main()** powoduje, że są to tzw. ***zmienne globalne*.**
Ciało funkcji **main()** jest nieskomplikowane:

```
13   int main()
14   {  clrscr();
15      info();
16      dane();
17      licz();
18      wyniki();
19      return 0;
20   }
```

W liniach 15, 16, 17 i 18 wywołujemy opracowane przez nas funkcje. Gdy program dojdzie do instrukcji **info(),** następuje wywołanie funkcji **info().** Sterowanie zostanie przekazane do tej funkcji. Wykonane są instrukcje umieszczone w ciele funkcji **info()**:

```
21   void info()
22   {  printf("\n%s\n%s",
23      "Uwaga: program wczytuje n liczb calkowitych",
24      " znajduje wartosc maksymalna");
25   }
```

W tych liniach umieszczona jest ***definicja*** funkcji **info().** Użyta w naszym programie funkcja **info()** powoduje umieszczenie następującego napisu na ekranie:

```
Uwaga: program wczytuje n liczb calkowitych
znajduje wartosc maksymalna
```

Po wykonaniu instrukcji zawartych w funkcji **info(),** sterowanie zostaje przekazane do funkcji **main()** i zostaje wykonana następna instrukcja. W naszym przypadku jest to wywołanie następnej funkcji - funkcji **dane().**
Po wykonaniu instrukcji zawartych w tej funkcji, sterowanie powraca do funkcji **main()** i następuje wywołanie funkcji **licz().**
Definicja tej funkcji ma postać:

```
33   int licz()
34   { max = x[0];
35      for (int i = 1; i<n; i++)
36         if (x[i]>max) max = x[i];
37      return max;
38   }
```

Zwróćmy uwagę, że funkcja **licz()** jest typu **int,** oznacza to, że zwracana wartość jest typu **int** (liczba całkowita). Funkcja **licz()** szuka maksymalnej liczby w tablicy **x[ ].** Instrukcja:

```
return max;
```

zwraca wartość **max,** w naszym przypadku jest to wartość największego elementu zmiennej **x[ ].** Zmienna **max** jest zmienną globalną, oznacza to, że każda inna funkcja może użyć tej zmiennej. W naszym przypadku następna funkcja **wyniki()** drukuje wartość **max** (największą liczbę spośród wprowadzonych do programu danych).

## 2.8. Wskaźniki

Wskaźniki są często traktowane, jako jedno z najtrudniejszych zagadnień języka C. Dzięki wskaźnikowi mamy zapewniony dostęp do zmiennej (lub innych skomplikowanych typów danych) bez bezpośredniego odwoływania się do tej zmiennej. Charakterystyczną cechą języka C i jego siłą jest stosowanie ***wskaźników*** i ***arytmetyki wskaźnikowej.***
Zmienna w programie jest przechowywana (zapamiętywana) w określonej ilości bitów w konkretnym miejscu w pamięci komputera. Mówimy o ***komórce pamięci***. Każda komórka pamięci ma swój ***adres***. Jest to zwykły numer (np. 61234). Jeżeli **zmien** jest zmienną, wtedy **&zmien** jest miejscem (lub adresem) w pamięci przechowującym jej wartość. **&** jest ***operatorem adresu***. Adresy są zbiorem wartości, którymi możemy manipulować tak, jak zwykłymi zmiennymi - mówimy o ***zmiennych adresowych*** lub krótko o ***wskaźnikach***. Wskaźnik jest zmienną, której wartością jest adres innej zmiennej. Do manipulacji wskaźnikami służą dwa specjalne operatory:

- operator adresu oznaczany jako & (ampersand)
- operator adresowania pośredniego * (gwiazdka)

W modelu pamięci operacyjnej, traktującym pamięć jako ciąg ponumerowanych komórek, adresem komórki jest jej numer. Zmienne wskaźnikowe mogą być deklarowane w programie a następnie wykorzystane do pobrania adresu. Deklaracja:

```
int *wsk;
```

deklaruje **wsk** jako wskaźnik do typu **int**.
Jeżeli w programie mamy zmienną **x** oraz zmienną wskaźnikową **px,** to wartość wyrażenia **&x** podaje adres tej zmiennej. Wartość tą (tutaj adres) możemy przypisać zmiennej wskaźnikowej **px**. Mówimy, że zmienna **px** jest wskaźnikiem zmiennej **x.**

W programie pokazanym na listingu 2.15 pokazujemy proste operowanie zmiennymi wskaźnikowymi. W programie mamy zdefiniowane dwie zmienne klasyczne **x** i **y** oraz dwie zmienne wskaźnikowe **wsk1** i **wsk2**. Program drukuje wartości zmiennych **x** i **y** oraz ich adresy (wartości zmiennych **wsk**1 i **wsk**2). W linii 09:

```
int *wsk1, *wsk2;
```

widzimy deklaracje zmiennych wskaźnikowych. Deklaracja zmiennej wskaźnikowej musi zawierać ***typ bazowy*** (tu mamy **int**), operator gwiazdki i nazwę (tutaj jest to **wsk1** i **wsk2**).

Listing 2.15. Wskaźniki.

```
01   /* C */
02   //  wskazniki, (ang. pointers)
03   #include <stdio.h>
04   #include <conio.h>
05   int main()
06   {
07     int x = 13;
08     int y = 33;
09     int *wsk1, *wsk2;
10     clrscr();
11     wsk1 = &x;
12     wsk2 = &y;
13     printf("\n x = %8d      y = %8d", x, y);
14     printf("\n &wsk1 = %8u  &wsk2 = %8u",
                  &wsk1,&wsk2);
15     printf("\n wsk1 = %8u   wsk2 = %8u",
                  wsk1,wsk2);
16     printf("\n Hex &wsk1 = %8x  &wsk2 = %8x",
                  &wsk1,&wsk2);
17     getche();
18     return 0;
19   }
```

Typ bazowy wskaźnika określa typ danych, na które wskaźnik może wskazywać. Poprawna deklaracja typu bazowego ma zasadnicze znaczenie. W liniach 11 i 12:

```
wsk1 = &x;
wsk2 = &y;
```

zmienna wskaźnikowa **wsk1** uzyskuje adres zmiennej **x**, a zmienna **wsk2** uzyskuje adres zmiennej **y**. W liniach 13 i 14:

```
printf("\n          x = %8d      y = %8d", x, y);
printf("\n     &wsk1 = %8u  &wsk2 = %8u",
&wsk1,&wsk2);
```

funkcja **printf()** powoduje wydruk wartości zmiennych **x** i **y** (linia 13) oraz adresów tych zmiennych. Często wartość adresu podawana jest w zapisie szesnastkowym. W linii 16:

```
printf("\n Hex &wsk1 = %8x  &wsk2 = %8x",
&wsk1,&wsk2);
```

pokazano, w jaki sposób drukujemy adres w systemie szesnastkowym - używając specyfikatora formatu:

```
%8x
```

Wskaźniki i tablice stosują prawie taki sam sposób, aby mieć dostęp do pamięci.

Są jednak różnice i to dość subtelne. Wskaźnik jest zmienną, której wartością jest adres. Nazwa tablicy jest adresem lub wskaźnikiem, który jest ustalony. Rozważmy deklarację:

```
int x[10], *px;
```

Dwie instrukcje:

```
px = x;
```
oraz
```
px = &x[0];
```

są równoważne.
W programie z listingu 2.16 ilustrujemy zastosowanie wskaźników. Program sumuje elementy tablicy. Pokazano dwa sposoby realizacji sumowania elementów tablicy.

Listing 2.16. Wskaźniki, sumowanie elementów tablicy.

```
01  /* C */
02  //  wskazniki i tablice
03  #include <stdio.h>
04  #include <conio.h>
05  int main()
06  {
07    int suma,  *px;
08    int x[10] = {1,2,3,4,5};
09    clrscr();
10    suma = 0;
11    for (px = x; px<&x[10]; ++px)
12      suma += *px;
13    printf("\n suma = %8d",suma);
14    suma = 0;
15    for (int i=0; i<10; ++i)
16      suma += *(x+i);
17    printf("\n suma = %8d",suma);
18    getche();
19    return 0;
20  }
```

W linii 07:

```
int suma,  *px;
```

zadeklarowano zmienną **suma** i zmienną wskaźnikową **px**. W linii 08:

```
int x[10] = {1,2,3,4,5};
```

mamy deklarację tablicy **x** oraz jej inicjalizację (nadanie wartości). Zauważmy, że tablica jest dziesięcioelementowa, pierwszych 5 elementów ma wartości pokazane w nawiasach klamrowych, pozostałe elementy maja wartość zero. W liniach 11 i 12

```
11 for (px = x; px<&x[10]; ++px)
12   suma += *px;
```

realizowane jest sumowanie elementów tablicy. W pętli, zmienna wskaźnikowa **px** jest inicjalizowana ***adresem bazowym*** tablicy **x**. Kolejne wartości **px** to **&x[1], &x[2]**,...itd. W linii 12 mamy instrukcję:

```
suma += *px;
```

W liniach 15 i 16:

```
15   for (int i=0; i<10; ++i)
16     suma += *(x+i);
```

pokazano inny sposób sumowania elementów tablicy.
Uzyskanie dostępu do tablic wielowymiarowych za pomocą wskaźników jest bardziej skomplikowane. Zakładając, że tablica **x** jest tablicą o wymiarach 10 na 10, dostęp do elementu 1,5 tablicy uzyskujemy przez jej indeksowanie, **x[1][5],** a za pomocą wskaźnika, ***((int *) x+5).** Podobnie, dostęp do elementu 1,2 otrzymamy indeksując element: **x[1][2],** gdy zechcemy skorzystać ze wskaźników musimy użyć zapisu: ***((int *) x + 12).** Te niuanse ilustruje program z listingu 2.17, który sumuje elementy dwuwymiarowej tablicy **x[ ][ ].**

Listing 2.17. Wskaźniki, tablice dwuwymiarowe.

```
/* C */
//  wskazniki i tablice wielowymiarowe
#include <stdio.h>
#include <conio.h>

int main()
{  int suma, *px;
   int x[2][3] = { 1,2,3,
                   4,5,6};
   suma = 0;
   px = (int*)x;
   for (int i=0; i<6; i++)
      suma += *(px+i);
   printf("\n suma = %8d",suma);
   getche();
   return 0;
}
```

W liniach:

```
int x[2][3] = {1,2,3,
               4,5,6};
```

mamy deklarację tablicy i jej inicjalizację. W linii:

```
px = (int*)x;
```

zastosowano jawną konwersje do typu **(int *).**

Pętla sumująca:

```
for (int i=0; i<6; i++)
   suma += *(px+i);
```

działa poprawnie

## 2.9. Struktury

Omawiane przez nas tablice są strukturą danych, dzięki którym mamy możliwość grupowania danych tego samego typu. W przypadku, gdy zachodzi konieczność grupowania danych różnych typów, język C dostarcza nam kolejną możliwość – tworzenie **struktur**. Wykorzystanie struktur omówimy na konkretnym przykładzie. Przypuśćmy, że chcemy skatalogować zbiór skryptów. Podstawowe informacje to : autor, tytuł i rok wydania**.** Potrzebujemy zmiennych różnych typów. Autor i tytuł zapisywane są w tablicy znakowej, rok wydania zapisujemy w zmiennej typu **int.** Struktura pozwoli na przechowanie danych różnych typów. Formalnie struktura jest zbiorem zmiennych, do których można się odwoływać korzystając z jednej nazwy. Deklaracja struktury jest wzorcem (szablonem) na podstawie, którego tworzymy egzemplarze struktury**.** W naszym przykładzie mamy jeden szablon opisu skryptu, możemy tworzyć opisy wielu skryptów. Zmienne**,** które wchodzą w skład struktury nazywają się składowymi struktury lub elementami struktury polami struktury. Na listingu 2.18 pokażemy jak tworzymy prostą strukturę i jak ją można wykorzystać. Przebieg działania programu może mieć postać:

```
Podaj autora:
Edward Angel
Podaj tytul:
OpenGL
Podaj rok wydania:
2008
Spis skryptow:
autor: Edward Angel, tytul : OpenGL, rok wydania: 2008
```

Deklaracja struktury pokazana na listingu 2.18 ma postać:

```
struct skrypt
{
   char autor[80];
   char tytul [80];
   int rok;
};
```

Listing 2.18. Struktury.

```
/*C*/
//struktury
#include <stdio.h>
#include <conio.h>
struct skrypt
{  char autor[80];
   char tytul [80];
   int rok;
};
int main()
{  struct skrypt spis;
   puts("Podaj autora :");
   gets(spis.autor);
   puts("Podaj tytul:");
   gets(spis.tytul);
   puts("Podaj rok wydania :");
   scanf("%d", &spis.rok);
   puts("Spis skryptow : ");
   printf("autor : %s,  tytul : %s, rok wydania : %8d",
            spis.autor, spis.tytul, spis.rok);
   getche();
   return 0;
}
```

Za pomocą słowa kluczowego **struct** informujemy kompilator, że tworzymy deklarację struktury. W pokazanej deklaracji mamy strukturę złożoną z dwóch tablic znakowych i jednej zmiennej typu **int**. Po deklaracji struktury umieszczamy średnik, ponieważ deklaracja struktury jest instrukcją. W deklaracji struktury występuje nazwa typu struktury, w naszym przypadku jest to literał (etykieta) **skrypt**. Nazwa **skrypt**, identyfikuje konkretny typ danych i ma znaczenie identyfikatora typu. Możemy utworzyć zmienną typu struktura:

```
struct skrypt spis;
```

Tego typu deklaracja informuje kompilator, że **spis** jest zmienną typu strukturalnego, o budowie szablonu **skrypt**. Wiele kompilatorów dopuszcza także formę deklaracji w postaci:

```
skrypt spis;
```

Jak widać można opuścić słowo kluczowe **struct**. Do poszczególnych składowych struktury możemy odwoływać się za pomocą operatora kropki ( **.** ), zostało to wykorzystane w instrukcjach:

```
gets(spis.autor);
gets(spis.tytul);
scanf("%d", &spis.rok);
```

Możemy bezpośrednio nadać wartość konkretnemu elementowi struktury.

Poniższy fragment kodu nadaje wartość 2005 polu **rok** struktury **skrypt**:

```
spis.rok = 2005;
```

Możemy tworzyć tablice struktur. W celu zadeklarowania tablicy struktur należy najpierw zdefiniować strukturę, a następnie należy zadeklarować zmienną tablicową wykorzystując typ strukturalny. Odwołujemy się do konkretnej struktury tablicy za pomocą indeksu (identycznie jak do prostej tablicy).

## 2.10. Zapis i odczyt plików

Wyniki otrzymywane z programów najczęściej kierowane są na ekran monitora – jest to bardzo użyteczna metoda prezentacji rezultatu wykonania programu. Należy jednak pamiętać, że po wyłączeniu komputera wyniki są bezpowrotnie tracone. Dane potrzebne do wykonania programu najczęściej wprowadzane są z klawiatury, – ale nie trudno sobie wyobrazić, że bardzo przydatne byłoby automatyczne wprowadzanie danych do programu z jakiegoś nośnika (jest to ważne, gdy np. dane pomiarowe z eksperymentu mogą być liczone w tysiącach). Program napisany w edytorze także musi być zapisany na trwałym nośniku.

Pamiętamy**,** że ***plikiem*** (ang. *file*) nazywamy ciąg bitów. Plik może być zapisany w pamięci zewnętrznej – np. na dysku twardym lub innym trwałym nośniku pamięci. Każdy plik identyfikowany jest przez nazwę. Język C udostępnia pokaźny zbiór funkcji do obsługiwania operacji plikowych.

Informacje możemy zapisać na dysku lub odczytać pod warunkiem, że plik jest otwarty – musimy po prostu zapewnić komunikację między systemem operacyjnym i programem. Sposób przesyłania danych z programu na trwały nośnik jest nieco skomplikowany. Program wysyła dane do specjalnego obszaru pamięci, zwanego ***buforem***. Z bufora dane są przenoszone na trwały nośnik. Użytkownik, w większości przypadków nie musi się martwić o bufor pamięci – jest on dla niego praktycznie „niewidoczny". Nowoczesne systemy operacyjne w zasadzie bezpośrednio nie obsługują urządzeń zewnętrznych takich jak klawiatura czy drukarka. Dostęp do tych urządzeń odbywa się za pomocą tzw**.** ***kanałów***. Program potrzebujący urządzenia zewnętrznego komunikuje się z kanałem, konkretny kanał powiązany jest z urządzeniem. Wszystkie operacje wejścia – wyjścia w języku C odwołują się do kanałów. Mamy następujące kanały:

- stdin – kanał dla danych wejściowych (*standard input*)
- stdout – kanał dla danych wyjściowych (*standard output*)
- stderr – kanał wyjściowy do obsługi komunikatów o błędach, ma standardowo przydzielony monitor, jako urządzenie zewnętrzne (*standard error*)

Jeżeli np. program wywołuje funkcję **getchar()** to ta funkcja nie przesyła znaku z klawiatury do pamięci. Przesłanie znaku do pamięci odbywa się z kanału, któremu system z góry przyporządkował klawiaturę. Oczywiście, język C oprócz standardowych kanałów dostarcza kilka dodatkowych mechanizmów

obsługi wejścia – wyjścia. Szczególnie użyteczne są funkcje biblioteczne obsługujące pliki dyskowe. Te funkcje automatyzują operacje na plikach tak, że programista nie musi zbytnio kłopotać się szczegółami technicznymi ( takimi jak np. rozmiar bufora, adresowanie itp.). Proces zapisu danych na dyskietkę wymaga współpracy programu i systemu operacyjnego. Podczas np. tworzenia pliku, program tworzy w pamięci specjalną strukturę, która przechowuje niezbędne informacje (potrzebne programowi i systemowi operacyjnemu). W języku C struktura jest kolekcją danych różnego typu. W deklaracji struktury, jak pamiętamy, korzystamy ze słowa kluczowego **struct**.
Kompilatory języka C przechowują informacje dotyczące plików w pliku nagłówkowym o nazwie **stdio.h**. Powstała struktura zdefiniowana jest, jako:

```
struct FILE
```

Każdy plik ma swoja strukturę FILE.
Pokażemy teraz prosty program służący do odczytu znaków z klawiatury, pobrane znaki będą zapisane w pliku dyskowym.

Listing 2.19. Zapis danych do pliku.

```
//czyta znaki i tworzy plik
#include <stdio.h>            //takze do obslugi pliku
#include <conio.h>            //funkcja getche()
int main()
{  FILE *fwsk; //wskaznik do FILE
   char znak;
   fwsk = fopen("plik_t1.txt","w");  // otwarcie pliku
   while ( (znak=getche()) != '\r')  // \r - ENTER
     putc(znak,fwsk);                 // zapis do pliku
   fclose(fwsk);                    // zamkniecie pliku
   return 0;
}
```

Aby można było zapisać dane w pliku należy taki plik „otworzyć". Jeżeli plik nie istnieje, zostanie automatycznie utworzony, (jeżeli nie będzie przeszkód).
Do otwierania pliku służy funkcja biblioteczna **fopen()**. Wywołanie tej funkcji ma postać:

```
idkan = fopen(nazwa_pliku, tryb)
```

| | | |
|---|---|---|
| `idkan` | – | identyfikator kanału |
| `nazwa_pliku` | – | nazwa pliku do otwarcia (np. " mojplik.txt") |
| `tryb` | – | informuje o akcji ( np. "r" – chcemy otworzyć plik tylko do czytania) |

Otwarty plik będzie miał unikalną strukturę zdefiniowana, jako **struct FILE** . Otwierając plik otrzymujemy wskaźnik do danej struktury FILE.
Funkcja **fopen()** zwraca wskaźnik do struktury FILE związanej z naszym plikiem, którą zapamiętuje w zmiennej **idkan.**

Pisząc program możemy zrobić to w następujący sposób.

1. Zaczynamy od funkcji **main()**:

   ```
   fwsk = fopen(”plik_t1.txt”, ”w”);
   ```

   W tej instrukcji informujemy kompilator, że chcemy otworzyć plik o nazwie **plik_t1.txt**, plik ma być otworzony do zapis (tryb **w**). Jeżeli plik nie istniej, będzie utworzony. Jeżeli plik o podanej nazwie już istnieje, poprzednie dane w nim zapisane będą utracone. Funkcja **fopen()** zwraca wskaźnik do struktury, wskaźnik zapamiętujemy w zmiennej **fwsk**.

2. Do wywołania funkcji **fopen()** potrzebujemy pliku nagłówkowego **<stdio.h>,** musimy także zadeklarować zmienną typu wskaźnik (zmienna wskaźnikowa **fwsk**), po wykonaniu zapisu w pliku, plik powinien być zamknięty (korzystamy z funkcji **fclose()**):

   ```
   FILE *fwsk;
   fwsk = fopen("plik_t1.txt","w");
   fclose(fwsk);
   ```

3. Możemy teraz zająć się tą częścią programu, która pobierze znak z klawiatury (zmienna **char znak,** funkcja **getche()**) zapisze go w pliku, zakończy działanie, gdy trafi na ogranicznik. Do czytania znaków wykorzystamy pętlę **while** i funkcję **putc()**. Działanie programu będzie zakończone po naciśnięciu klawisza Enter.

   ```
   char znak ;
   while( znak = getche() != ’\r’ )
     putc( znak, fwsk ) ;
   ```

4. Po uruchomieniu, program czeka na wpisanie tekstu. Po naciśnięciu klawisza Enter, program kończy działanie. Możemy sprawdzić czy utworzony został na dysku plik o nazwie **plik_t1.txt** i co zawiera.

# ROZDZIAŁ 3

## PODSTAWOWE TYPY DANYCH, STAŁE, ZMIENNE, WYRAŻENIA, INSTRUKCJE, OPERATORY

## 3.1. Wstęp

Opis języka C zaczniemy od przedstawienia najbardziej podstawowych elementów. Omówimy typy danych, zmienne i stałe, wyrażenia, instrukcje i operatory. W zależności od indywidualnego spojrzenia na język C, autorzy podręczników programowania wyróżniają następujące podstawowe typy danych:

- znak
- liczba całkowita
- liczba zmiennopozycyjna typu float
- liczba zmiennopozycyjna typu double
- brak wartości (void)
- bool

Niestety nie został ustalony standard zapisu podstawowych typów danych (np. typ **int** może być zapisywany na dwóch lub czterech bajtach ). Ustalenia ilości bajtów, wymaganych dla konkretnego kompilatora należy do programisty. Należy także pamiętać, że język C zbyt mocno nie wspiera obsługi napisów. Formalnie napis jest tablicą znakową.

W języku C program jest sekwencją znaków, które kompilator języka przekształca do postaci zrozumiałej dla konkretnego komputera. Pisząc program korzystamy z elementów języka C. Program musi być napisany poprawnie językowo. Do podstawowych elementów języka należą:

- zestaw znaków
- nazwy i słowa kluczowe (zastrzeżone)
- typy danych
- stałe, zmienne i tablice
- definicje i deklaracje
- wyrażenia
- instrukcje

Programy działają w oparciu o dane. Wprowadzając do komputera liczby, litery czy słowa, oczekujemy, że zostaną one w odpowiedni sposób przetworzone. Pisząc program musimy określić, z jakiego typu danych będziemy korzystać.

## 3.2. Zestaw znaków, słowa kluczowe

Jak każdy język, język C ustala zestaw znaków (alfabet), którymi można się posługiwać. Oto zestaw znaków języka C:

1 26 wielkich liter alfabetu łacińskiego:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

2 26 małych liter alfabetu łacińskiego:

a b c d e f g h i j k l m n o p q r s t u v x y z

3 10 cyfr:

1 2 3 4 5 6 7 8 9 0

4 31 symboli wprowadzanych z klawiatury (znaki specjalne) ;

! @ # $ % ^ & * ( ) - _ + = { [ } ] : ; ‘ „ < , > . ? / | \ `

5 znak odstępu (spacja)

Słowa kluczowe stanowią słownictwo języka C. Ponieważ mają specjalne znaczenie, nie można używać ich, jako nazw zmiennych. W zasadzie, słowa kluczowe objęte są standardem, który w praktyce nie jest przestrzegany, wiele współczesnych kompilatorów języka C wprowadza nowe słowa kluczowe (np. **asm** czy **pascal**), dlatego warto sprawdzić zestaw słów kluczowych pracując z konkretnym kompilatorem.

Tabela 3.1 Słowa kluczowe języka C (standard ANSI C oraz standard C9X)

| | | | |
|---|---|---|---|
| auto | double | inline | static |
| break | else | int | struct |
| case | enum | long | switch |
| char | extern | register | typedef |
| complex | float | restrict | union |
| const | for | return | unsigned |
| continue | goto | short | void |
| default | if | signed | volatile |
| do | imaginary | sizeof | while |

## 3.3. Zasady leksykalne języka C

Program napisany w języku C jest zbiorem znaków. Kompilator języka C przekształca tak napisany kod na ***program wykonywalny***, który może być uruchamiany. Aby kompilacja przebiegła prawidłowo, program musi być napisany prawidłowo pod względem syntaktycznym. Kompilator sprawdza poprawność językową programu. Znaki są zbierane (grupowane w łańcuchy) w tzw. ***tokeny*** (ang. ***tokens***), które muszą być zgodne z zasadami języka. Tokeny traktujemy, jako zbiór zasobów języka. Mówiąc obrazowo, kompilator sprawdza poprawność słownictwa, gramatyki i pisowni.

W porównaniu do języków naturalnych (np. polskiego, angielskiego, chińskiego) język komputerowy jest bardzo restrykcyjny – nie toleruje żadnych odstępstw od przyjętych reguł.

Znaki są grupowane przez kompilator w syntaktyczne jednostki, którymi są identyfikatory, słowa kluczowe, stałe, łańcuchy, operatory i inne separatory. Omówimy te pojęcia analizując typowy program.
W programie występują ***ograniczniki*** (ang. *delimiter*) takie jak

```
/*  ..... */    czy    //
```

Są to komentarze, podczas kompilacji nie są one brane pod uwagę, kompilator ograniczniki zastępuje odstępem (ang. ***white space***) .
W zapisie:

```
main()
```

nazwa funkcji **main** jest ***identyfikatorem***, znaki nawiasów **(** oraz **)** są ***separatorami***.
W linii

```
int ax, bx, suma ;
```

**int** jest słowem kluczowym, zmienne **ax**, **bx** i **suma** są identyfikatorami, znak przecinka ” **,** ” oraz średnika ” **;** ” są separatorami.
Jeżeli mamy zapis:

```
scanf (”%d%d”,  &ax, &bx) ;
```

to nazwa funkcji **scanf** jest identyfikatorem, występujące w tym zapisie nawiasy **(** oraz **)** informują kompilator, że **scanf()** jest funkcją, (tak samo jak **main**) .
Występujący w tej linii ciąg znaków:

```
”%d%d”,
```

jest stałym łańcuchem. W zapisie

```
&ax   oraz    &bx
```

znak **&** jest ***operatorem***.
W zapisie:

```
suma = ax + bx;
```

symbole ” + ” oraz ” = ” są także operatorami. Odstępy w tym zapisie są ignorowane, np. następujące zapisy są równoważne:

```
suma = ax + bx;
suma=ax+bx;
suma   =    ax   +    bx;
```

natomiast nie można wstawiać spacji w identyfikatorze, zapis:

```
s u m a = ax + bx ;
```

jest niepoprawny !
Należy pamiętać, że w języku C ten sam symbol może mieć różne znaczenia, w zależności od kontekstu. Powyżej symbol **%**, występujący w zapisie :

```
scanf (”%d%d”,  &ax, &bx) ;
```

oznacza tzw. ***specyfikator formatu***, a w zapisie :

```
ilocz = ax % bx ;
```

symbol ” **%** ” jest interpretowany, jako ***operator dzielenia modulo***.
Kodując program musimy specjalną uwagę zwracać na sposób, w jaki kompilator dokonuje analizy leksykalnej tworząc tokeny. Rozpatrzmy np. następujący zapis:

```
ax+++bx
```

Wiemy, że symbol ” + ”jest operatorem dodawania, symbol ” ++ ” jest ***operatorem inkrementacji***. Znaczenie tego zapisu może być następujące:

```
ax++    +    bx
```

lub

```
ax      +      ++bx
```

Oczywiście każdy typ kompilatora ma precyzyjnie reguły na interpretacje tego typu zapisów, ale musimy być pewni takich reguł. Gdy są jakiekolwiek wątpliwości należy stosować nawiasy.
**Identyfikatorem** nazywamy sekwencję liter, znaków i symbolu ” _ ”. Identyfikator musi rozpoczynać się od litery albo od symbolu ” _ ” . Nie może zawierać odstępu lub znaku specjalnego. Wybierając identyfikator staramy się o nazwę mającą znaczenie mnemotechniczne, tak jak w przykładach:

```
pole = bok_a * bok_b
suma_wag = waga1 + waga2
```

Oczywiście możemy napisać:

```
z = x * y
w = a + b
```

Przykłady poprawnych identyfikatorów:

```
ax
_nazwisko
student121
moje_dane
mojeDane
```

Błędne są następujące identyfikatory:

| | |
|---|---|
| `paragraf#01` | symbol specjalny # nie jest dozwolony |
| `1999rok` | identyfikator nie może rozpoczynać się od cyfry |
| `-dane` | typowy błąd, zamiast znaku _ napisano znak – |
| `nr katalog` | w nazwie nie może być spacji |

Jako identyfikatorów nie możemy stosować słów kluczowych. Litery małe są innymi znakami niż duże, identyfikator **abc** jest różny od **Abc**, tak samo jak **Abc** czy **ABC**.

Ważna jest długość identyfikatora – niektóre kompilatory dopuszczają 8 znaków w identyfikatorze, inne dużo więcej (np. 80).

W praktyce programowania przyjęły się pewne zasady nadawania nazw zmiennym:

- stosuje się wyłącznie małe litery, z wyjątkiem stałych określonych za pomocą dyrektywy preprocesora **#define** oraz nazw typów określonych za pomocą **typedef**
- używa się litery **c** do oznaczenia zmiennych znakowych
- używa się litery **s** do znaczenia ciągów znaków (łańcuchów)

## 3.4. Definicje i deklaracje

Rozróżnia się pojęcie deklaracji od pojęcia definicji. Są to ważne pojęcia, w praktyce często się o tym zapomina.

***Deklaracja zmiennej*** określa nazwę i typ zmiennej (nie rezerwuje pamięci).

***Definicja zmiennej*** określa nazwę, typ zmiennej oraz rezerwuje miejsce w pamięci komputera.

Deklaracje zmiennych są bardzo istotne w programach, które wykorzystują wiele plików - często zachodzi przypadek, że zmienna jest zdefiniowana w jednym pliku a następnie musi być używana przez inny plik.
***Zmienne*** i ***stałe*** są obiektami, na których program wykonuje niezbędne operacje. Deklaracje służą do przyporządkowania zmiennym odpowiedniego typu. Można nadawać wartość początkową w deklaracji. W zasadzie wszystkie zmienne muszą być zadeklarowane przed użyciem.
Jak już wiemy, podstawowe typy danych to: **int, char, float, double,** łącznie z modyfikatorami: **short, long, signed, unsigned.** Podstawowe typy danych łącznie z modyfikatorami dostarczają nowych typów danych.
Oprócz podstawowych typów danych mamy także inne typy danych:

- tablice
- struktury
- wskaźniki
- unie
- dane wyliczeniowe (**enum**)
- łańcuchy (w zasadzie tablice)
- dane zdefiniowane za pomocą specyfikatora **typedef**

Metody posługiwania się tymi typami danych będą omówione w dalszej części podręcznika.

## 3.5. Typy danych

Język programowania, jakim jest C inaczej obsługuje dane, które są liczbami całkowitymi a inaczej dane, które są liczbami rzeczywistymi. W języku C mamy

kilka podstawowych typów danych. Podstawowy zestaw elementarnych typów danych języka C to:

- int — reprezentuje liczbę całkowitą
- char — reprezentuje pojedynczy znak
- float — reprezentuje liczbę rzeczywistą
- double — reprezentuje liczbę rzeczywistą o podwójnej precyzji

Wśród wielu powodów wprowadzenia do języka C różnych typów danych, jednym z głównych było dążenie do oszczędnego gospodarowania pamięcią.
Standard K&R języka C (ustalony przez Briana Kernighana i Dennisa Ritchie'go) wykorzystuje siedem słów kluczowych określających typy:

- int
- long
- short
- unsigned
- char
- float
- double

Standard ANSI C dodał jeszcze cztery słowa kluczowe:

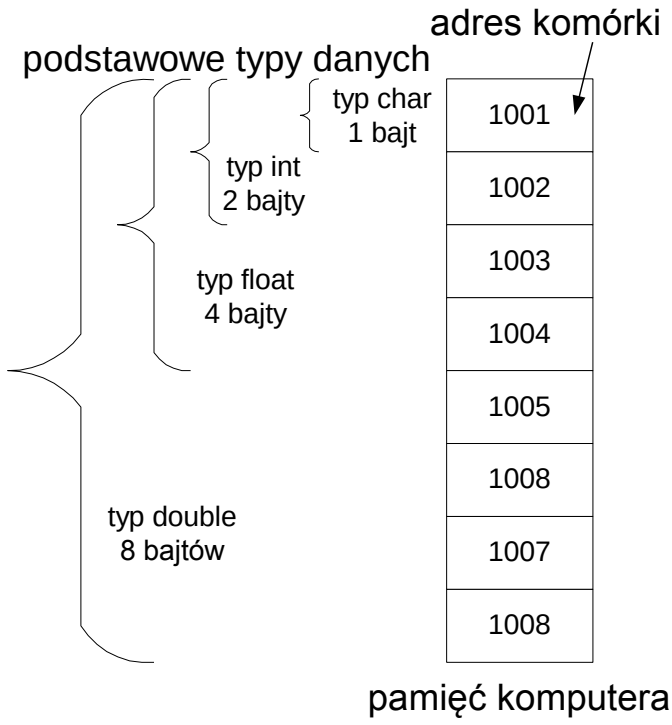
- signed
- void
- const
- volatile

Liczby są fundamentalnymi danymi używanymi przez komputery. Pamięć komputera (funkcjonalnie) zawiera liczby binarne. Te liczby grupowane są w sekwencje 8 bitów (**1 bajt**), 16 bitów (**2 bajty**), itd. W zależności od konkretnych potrzeb możemy chcieć operować na liczbach 1-bajtowych czy też np. 10-bajtowych. Kompilator musi znać typ danej, oby przydzielić odpowiednią liczbę bitów (bajtów) w pamięci. Liczby przechowywane są w ***komórkach***, każda komórka ma ***adres***. Model pamięci (w realizacji 1-bajtowej) przedstawiony jest na rys. 3.1.

W języku C zmienne służą do przechowywania informacji. Do tego potrzebne jest miejsce w pamięci komputera. Pamięć komputera możemy traktować, jako zbiór pojemników. Każdy taki pojemnik składa się z 8 bitów, każdy bit może być „ustawiony", tzn. może mieć wartość 1 lub wartość 0. Każdy pojemnik, czyli miejsce w pamięci, posiada indywidualny numer – ***adres pamięci***. Jak już powiedzieliśmy, zmienna rezerwuje miejsce w pamięci na przechowywanie wartości. Do tego może potrzebować jednej komórki lub też więcej komórek. Np. zmienną typu **float** może potrzebować 4 bajtów, czyli 32 bitów.

W praktyce okazuje się, że te podstawowe cztery typy danych nie pozwalają na zbyt racjonalne wykorzystanie pamięci. W języku C mamy dodatkowe ***modyfikatory***:

- short (liczba krótka)
- long (liczba długa)
- signed (liczba ze znakiem)
- unsigned (liczba bez znaku)




Rys.3.1 Model pamięci, jeden bajt składa się z 8 bitów.

Możliwy do zadeklarowania zestaw typów danych jest znacznie rozszerzony. Zależy on od typu kompilatora. W tabeli 3.1 podano typy danych podstawowych dla starszego kompilatora Borland Turbo C++. Dane te należy traktować, jako dane dydaktyczne, ponieważ co roku do sprzedaży wchodzą coraz mocniejsze procesory (wielobajtowe).

Jak już powiedzieliśmy, pamięć komputera podzielona jest na bajty, które tworzą komórki pamięci. Są one ponumerowane. Numerowanie przebiega od 0 do górnej granicy pamięci. Te numery są adresami bajtów pamięci. Dane umieszczane są kolejno w pamięci - jak wiemy różne typy danych potrzebują różnej ilości bajtów. Adres konkretnej danej jest adresem pierwszego bajta,

który ta dana zajmuje. W języku C stosunkowo łatwo możemy otrzymać numer adresu i rozmiar danej. Potrzebne są nam dwa operatory:

- operator adresu **&**
- operator rozmiaru **sizeof**

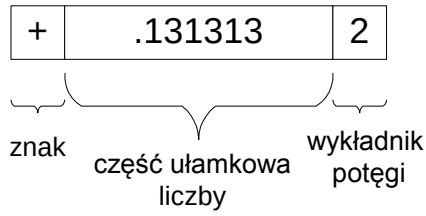
Tabela 3.2. Typy danych.

| Typ | Rozmiar (w bitach) | Zakres wartości | Typowe zastosowania |
|---|---|---|---|
| unsigned char | 8 | od 0 do 255 | małe liczby pełny zestaw znaków ASCII na PC |
| char | 8 | od -128 do 127 | bardzo małe liczby i zestaw znaków ASCII |
| enum | 16 | od -32768 do 32767 | uporządkowany zbiór wartości (inaczej - typ wyliczeniowy) |
| unsigned int | 16 | od 0 do 65535 | duże liczby i kontrola pętli |
| short int | 16 | od -32768 do 32767 | małe liczby kontrola pętli |
| int | 32 | od -2147483648 do 2147483647 | liczby, kontrola pętli |
| unsigned long | 32 | od 0 do 4294967295 | duże liczby |
| long | 32 | od -2147483648 do 2147483647 | duże liczby |
| float | 32 | od $3.4x10^{-38}$ do $3.4x10^{38}$ | obliczenia naukowe, dokładność do 7 znaków po przecinku |
| double | 64 | od $1.7x10^{-308}$ do $1.7x10^{308}$ | obliczenia naukowe, dokładność do 15 znaków po przecinku |
| long double | 80 | od $3.4x10^{-4932}$ do $1.1x10^{4932}$ | obliczenia finansowe i naukowe |

## 3.6. Liczby całkowite i zmiennoprzecinkowe

Liczby całkowite i liczby rzeczywiste są traktowane odmiennie. ***Liczba całkowita*** jest liczbą nieposiadającą części ułamkowej. ***Liczba zmienno-przecinkowa*** jest komputerowym odpowiednikiem liczby rzeczywistej.
Zapisując liczbę całkowitą w pliku źródłowym nigdy nie używamy kropki dziesiętnej. Przykłady poprawnie napisanych liczb całkowitych to: -15, 1313 czy 111111.
Liczby zmiennoprzecinkowe zapisujemy wykorzystując kropkę dziesiętną: 5.55 czy -13.13. Zapis: 5.0 oznacza liczbę zmiennoprzecinkową a nie liczbę całkowitą „pięć”. Liczby zmiennoprzecinkowe można zapisywać wykorzystując notację wykładniczą, np. 3.13E12.

W systemach komputerowych liczby zmiennoprzecinkowe są przechowywane w specyficzny sposób – zapisywany jest znak, część ułamkowa i wykładnik potęgi. Tego typu zapis liczby w postaci dziesiętnej pokazany jest na rys. 3.2.




$$+\ 0.131313E2 = +\ 0.131313x10^{2} = +\ 13.1313$$

Rys.3.2. Komputerowa postać liczby rzeczywistej (zmiennoprzecinkowej).

Istnieją języki programowania, które stosują jedynie liczby rzeczywiste. Korzystając z liczb całkowitych i zmiennoprzecinkowych używanych w języku C musimy pamiętać o następujących ograniczeniach:

- zakres liczb całkowitych jest stosunkowo nieduży, zakres liczb zmiennoprzecinkowy jest praktycznie wystarczający do wszelkich obliczeń
- wykonując pewne operacje na liczbach zmiennoprzecinkowych (np. odejmowanie) w pewnych przypadkach jesteśmy narażeni na utratę dokładności
- nieskończony zbiór liczb rzeczywistych ma w komputerze jedynie ograniczoną, dyskretną reprezentację liczb zmiennoprzecinkowych. Wartości zmiennoprzecinkowe są najczęściej przybliżeniami liczb rzeczywistych
- reprezentacje binarne ułamków dziesiętnych mogą prowadzić do ułamka okresowego
- liczby zmiennoprzecinkowe wymagają zdecydowanie więcej pamięci a działania na nich wykonują się znacznie wolniej niż na liczbach całkowitych

Projektując działania na liczbach zmiennoprzecinkowych należy kierować się rozsądkiem, w przypadkach wątpliwych należy program testować. Jako ciekawostkę zanalizujemy banalny przypadek. Jeżeli weźmiemy dowolna liczbę wyjściową, dodamy do niej jeden a następnie od otrzymanego wyniku odejmiemy naszą liczbę wyjściową to powinniśmy otrzymać jeden:

```
y = a + 1
x  = y – a
```

Sprawdzimy, jak tego typu działanie realizowane jest w praktyce. Przykładowy program realizujący dodawanie i odejmowanie może mieć postać pokazaną na listingu 3.2.

Listing 3.2 Działania arytmetyczne na dużych liczbach zmienno-przecinkowych.

```
//bledy operacji arytmetycznych
#include <stdio.h>
#include <conio.h>
int main()
{ float x,y;
  y = 5.0e5 + 1.0;
  x = y - 5.0e5;
  printf("%f \n", x);
  getch();
  return 0;
}
```

Po wykonaniu programu otrzymamy wydruk:

```
1.000000
```

co jest zgodne z naszymi oczekiwaniami.
Do jedynki chcemy dodać większą liczbę, odpowiedni fragment programu ma postać:

```
y = 5.0e10 + 1.0;
x = y - 5.0e10;
```

Po wykonaniu programu otrzymujemy wynik:

```
-1024.000000
```

Operacja dodawania i odejmowania została wykonana nieprawidłowo. Otrzymany wynik zależy od kompilatora, nasz wynik otrzymano na platformie Borland C 3.1. Komputer przechowuje ustaloną ilość cyfr znaczących. Zazwyczaj jest to 6 lub 7 cyfr. Dodając do liczby 5.0e5 (500 000) jedynkę, zmieniamy szóstą cyfrę, kompilator jest w stanie obsłużyć taką operację. Gdy do liczby 5.0e15 (ta liczba to 5 z piętnastoma zerami) chcemy dodać jedynkę próbujemy zmienić szesnastą cyfrę. Aby przeprowadzić taką operację kompilator powinien mieć możliwość zapisywania liczby składającej się z szesnastu cyfr. Jak już wiemy typ **float** może przechowywać najwyżej 6,7 cyfr znaczących. W takim przypadku otrzymamy (bez ostrzeżenia) niepoprawny wynik. Tego typu błędy są trudne do wykrywania. Zmiana typu danej na typ **double** pozwala wykonywać obliczenia z większą precyzją, zazwyczaj jest to, co najmniej 10 cyfr znaczących. Wykonując operacje arytmetyczne na liczbach zmiennoprzecinkowych można spowodować tzw. ***przepełnienie*** lub ***niedomiar*** wartości zmiennoprzecinkowej. Przepełnienie powstaje, gdy zostaje przekroczony zakres wartości. Np. liczbę typu **float** równą 1.0e38 mnożymy przez 1000.0 (też typu **float**). Niedomiar powstaje przy próbie dzielenia liczby typu **float,** np. 1.0e-37 przez dużą liczbę, też typu **float**, np. 1.0e8. Wynik takich operacji zależy od komputera. Może być nim przerwanie pracy programu i wyświetlenie komunikatu o błędzie wykonani (ang. run-time terror).

## 3.7. Stałe i zmienne

Tworząc program, używamy danych. Każdy element danych musi być zaklasyfikowany albo, jako ***stała*** albo, jako ***zmienna***.
***Stała*** pozostaje taka sama (nie może się zmieniać) w trakcie wykonywania programu – nadaje się jej ***wartość*** w trakcie tworzenia programu.
***Zmienna*** może otrzymywać różne wartości w trakcie wykonywania programu – wartość przypisywana jest podczas wykonywania programu.

Stała wartość musi być określonego typu. W języku C mamy cztery rodzaje stałych, przedstawionych w tabeli 3.3.

Tabela 3.3. Typy stałych.

| Rodzaj | Przykłady |
|---|---|
| Stałe całkowitoliczbowe | 1234 (dziesiętnie)<br>0377 (ósemkowo)<br>0x2f (szesnastkowo) |
| Stałe rzeczywiste | 3.14 3.14E4 |
| Stałe znakowe | ‘a’ ‘b’ ‘c’ ‘d’ |
| Łańcuchy znaków | ” abcd” |

W języku C możemy stosować następujące stałe całkowite:

- stałe dziesiętne ( system liczbowy dziesiętny)
- stałe ósemkowe (system liczbowy ósemkowy)
- stałe szesnastkowe (system liczbowy szesnastkowy)

Stosując stałe całkowite musimy przestrzegać odpowiednich reguł. Podstawową sprawą jest spełnienie warunku, że wartość stałej nie może wykraczać poza przedział dozwolonych dla danego kompilatora wartości.
Do zapisu stałych dziesiętnych korzystamy ze zestawu znaków:

```
+ - 1 2 3 4 5 6 7 8 9 0
```

Taka liczba nie może zaczynać się od znaku 0. Przykłady zapisu:

```
1947   -13    +3333
```

Do zapisu stałych ósemkowych (oktalnych) korzystamy ze zestawu znaków:

```
+ - 1 2 3 4 5 6 7 0
```

W poprawnym zapisie ósemkowym pierwszym znakiem **musi** być znak 0 (zero). Przykłady zapisu:

```
01747     -013      +03333
```

Do zapisu stałych szesnastkowych (zwanych też heksadecymalnymi) korzystamy ze zestawu znaków:

```
+ - 1 2 3 4 5 6 7 8 9 0 a b c d e f A B C D E F
```

W poprawnym zapisie szesnastkowym pierwszym znakiem **musi** być 0x lub 0X (zero – iks). Przykłady zapisu:

```
0x13        0xabc 0xffff
```

Przypominamy, że bez względu na sposób zapisu, liczby zapisywane są w postaci binarnej. Możemy w kodzie źródłowym umieścić liczby np. 16, 020 lub 0X10, a w każdym przypadku wartości będą przechowywane w postaci liczby binarnej. Zapis szesnastkowy czy ósemkowy został wprowadzony jedynie dla wygody programisty – nie ma wpływu na przebieg obliczeń czy wydajność.
Stałe rzeczywiste reprezentują liczby z systemu dziesiętnego. Formalnie liczby te nazywane są liczbami zmiennoprzecinkowymi.
Do zapisu stałych rzeczywistych korzystamy ze zestawu znaków:

```
+ - .  e E 1 2 3 4 5 6 7 8 9 0
```

Przykłady zapisu:

```
13.13   0.0007   -22.33          13.2e-3   4E4
```

Litera e (lub E) pozwala na tzw. „zapis naukowy”. W takiej notacji liczba np.:

$3.3 \times 10^{-5}$

jest zapisywana jako:

```
3.3e-5
```

Typ stałej całkowitej czy rzeczywistej zależy od jej postaci, wartości i przyrostka. Dla liczby całkowitej dziesiętnej typ domyślny jest typem **int, long int** lub **unsigned long int**. Dla liczb ósemkowych i szesnastkowych jest typem **int, unsigned int, long int, unsigned long int**. Jeżeli dodamy specyfikator **u** (lub **U**) a także **l** (lub **L**) wymuszamy zmianę sposobu reprezentowania stałej całkowitej.
Przykładem jest:

```
13UL            133u            4444L
```

Pierwsza liczba jest typu **unsigned long int,** druga jest **unsigned int** , trzecia jest **long int.** Domyślnym typem dla liczb rzeczywistych jest **double**. Specyfikator **f** (lub **F**) oraz **l** (lub **L**) wymusza typ **float** albo **long double.**
Przykładem jest:

```
-13.1f      1.1L
```

Pierwsza liczba jest typu **float**, druga typu **long double**. Liczby kodowane są na różnej ilości bajtów, zależy to od konkretnego kompilatora.
Praktyczny program do sprawdzania rozmiaru typów pokazany jest na listingu 3.3. Wykorzystany został operator **sizeof**, który zwraca rozmiary zmiennych i typów w bajtach.

Listing 3.3. Rozmiary typów, operator sizeof().

```
//sprawdza rozmiar typu
#include <stdio.h>
#include <conio.h>
int main()
{ for (int i=1; i<40; i++) printf("\xC4");
  printf ("\ntyp  rozmiar (bajty)  rozmiar (bity)\n");
  for (int j=1; j<40; j++) printf("\xC4");
  printf("\nchar \t\t %d  \t\t   %d\n",
    sizeof(char),8*sizeof(char));
  printf("int \t\t %d  \t\t   %d\n",
    sizeof(int),8*sizeof(int));
  printf("short \t\t %d  \t\t   %d\n",
    sizeof(short),8*sizeof(short));
  printf("long \t\t %d  \t\t   %d\n",
    sizeof(long),8*sizeof(long));
  printf("float \t\t %d  \t\t   %d\n",
    sizeof(float),8*sizeof(float));
  printf("double \t\t %d  \t\t   %d\n",
    sizeof(double),8*sizeof(double));
  printf("long double \t %d  \t\t   %d\n",
    sizeof(long double),8*sizeof(long double));
  getche();
  return 0;
}
```

Funkcja **printf()** do wyświetlanie wartości wymaga odpowiedniego specyfikatora formatu. Aby wyświetlić np. liczbę typu **unsigned int** korzystamy ze specyfikatora **%u**. Aby wyświetlić liczbę typu **long** wykorzystujemy specyfikator **%ld.** Specyfikator **%lx** wyświetla wartość liczby **long int** w postaci szesnastkowej. Specyfikator **%hd** wyświetla liczbę typu **short int** w postaci dziesiętnej a specyfikator **%ho** w postaci ósemkowej. Ilość kombinacji jest bardzo duża, może to prowadzić to kłopotów. Użycie niewłaściwego specyfikatora formatu może dać nieoczekiwane wyniki.
Przykładowy wynik działania programu pokazanego na listingu 3.3 wygląda następująco:

| Typ | rozmiar (bajty) | rozmiar (bity) |
|---|---|---|
| char | 1 | 8 |
| int | 2 | 16 |
| short | 2 | 16 |
| long | 4 | 32 |
| float | 4 | 32 |
| double | 8 | 64 |
| long double | 10 | 80 |

Stałymi znakowymi są pojedyncze znaki ujęte w pojedyncze apostrofy. Zestawem dozwolonych znaków jest zestaw znaków kodu ASCII.

Mamy 128 znaków kodu ASCII (wartości od 0 do 127) a także tzw. rozszerzony kod ASCII (wartości od 128 do 255).
Przykładem może być:

```
'x'     '%'     'A'
```

Do stałych znakowych zaliczamy także znaki niedrukowalne (tzw. sekwencje ucieczki, ang. *escape sequences*). Sekwencje ucieczki rozpoznawane są za pomocą znaku \ (backslah) i litery. Przykładem takiego znaku jest np**. \n** – nowa linia. Sposób używania znaków niedrukowalnych w języku C jest nieco zagmatwany. Zapis **\a** oznacza alarm (ten znak powoduje generowanie dźwięku przez głośnik komputera). Zestaw sekwencji sterujących przedstawiony jest w tabeli 3.4.

Tabela 3.4 Sekwencje sterujące (znaki niedrukowalne).

| Znak | Opis |
|---|---|
| \a | Alarm (dźwięk z głośnika) |
| \b | Znak cofania ( backspace) |
| \f | Wysuniecie strony ( form feed) |
| \n | Nowa linia |
| \r | Powrót karetki (carriage return) |
| \t | Tabulator poziomy |
| \v | Tabulator pionowy |
| \\ | Lewy ukośnik |
| \’ | Apostrof |
| \” | Cudzysłów |
| \0xx | Wartość ósemkowa (x oznacza cyfrę ósemkową) |
| \xhh | Wartość szesnastkowa (h oznacza cyfrę szesnastkową) |

Wartość ASCII znaku „alarm” wynosi 7, możemy stosować zapis:

```
char glosnik = 7;
```

Typowe użycie znaku niedrukowalnego to użycie apostrofu:

```
nowa_linia = '\n';
```

Różne kompilatory reagują różnie na znaki ucieczki. Jeżeli kompilator nie rozpozna znaku alarm (**\a**), można spróbować następującego zapisu:

```
glosnik = '\007'
```

Zera poprzedzające cyfrę 7 można pominąć, wystarczy napisać ‘\07’ lub ‘\7’. Taki zapis sugeruje, że liczby są ósemkowe, mimo, że nie ma poprzedzającego zera.
Zapamiętanie kodów ASCII jest dość kłopotliwe. Możemy napisać prosty program wyświetlający tabelę kodów ASCII.

Interpretując taka tabele musimy znać osobliwości kodów. Na przykład wiemy, że kody o wartościach dziesiętnych od 0 do 31 są tzw. kodami sterującymi. Przedstawiony program wyświetla tabelę kodów ASCII w podanym przez użytkownika zakresie wartości. Wyświetlana jest wartość kodu dziesiętnie, szesnastkowo oraz pokazywany jest znak.

Listing 3.4 Program do przedstawienia kodów ASCII.

```
/* kody ASCII */
#include <stdio.h>
#include <conio.h>
int main()
{ int n, start, koniec;
  printf("\npodaj zakres wartosci ASCII np:32 128\n");
  scanf("%d %d",&start,&koniec);
  printf("\n       Kody ASCII  w zapisie :  \n");
  printf(" wartosc dziesietna=szesnastkowa= znak\n");
  for (n=start ; n<koniec ; n++)
    printf("|%4d = %x = %c\t",n,n,n);
  getch();
  return 0;
}
```

Zasadniczą część programu wykonuje pętla **for** oraz funkcja **printf()**:

```
for (n=start ; n<koniec ; n++)
    printf("|%4d = %x = %c\t",n,n,n);
```

Do wprowadzenia zakresu wykorzystano funkcję **scanf(),** która wczytuje wprowadzone z klawiatury wartości zmiennych **start** i **koniec.** W funkcji **printf()** zastosowano trzy specyfikatory formatu:

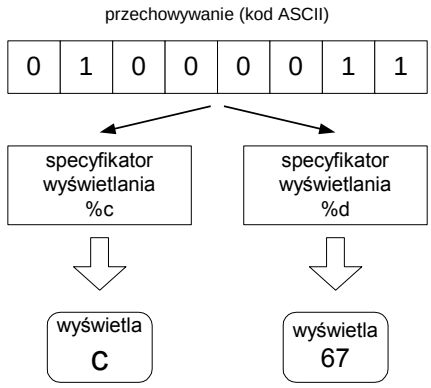
%d – wyświetla liczby całkowite
%x – przekształca liczbę typu **int** na postać szesnastkową i wyświetla
%c – wyświetla znak (typ **char**)

Możemy także zastosować specyfikator **%o** (mała litera o);

%o - przekształca liczbę typu **int** na postać ósemkową i wyświetla

Polecenie **\t** przenosi kursor do następnego punktu tabulacji, dzięki temu mamy bardziej estetyczny wydruk. Funkcja **getch()** służy do „przytrzymania" ekranu. Stosowanie kodów szesnastkowych ma tą zaletę, że zapis kodu jest krótki, zapis ósemkowy jest standardem – wszystkie komputery muszą rozpoznawać ten kod. Funkcja **printf()** do wyświetlania znaków korzysta ze specyfikatora **%c**. Zmienna znakowa jest przechowywana, jako jedno-bajtowa wartość całkowita. Jeżeli zmienna **char** będzie wyświetlana za pomocą funkcji **printf()** ze specyfikatorem **%d**, otrzymamy liczbę całkowitą.

Specyfikator **%c** zleca funkcji **printf()** zamianę liczby na odpowiadający jej znak. Należy pamiętać, że specyfikatory funkcji **printf()** określają sposób wyświetlania danych, a nie sposób ich przechowywania. Należy rozróżniać przechowywanie danych i ich wyświetlanie. To rozróżnienie zilustrowane jest na rys.3.3.
Należy tez pamiętać, że w niektórych kompilatorach języka C, typ **char** jest typem, który posiada znak, ten typ może przechowywać wartości z zakresu -128 do +127. W innych kompilatorach, jest typem bez znaku, ten typ może przechowywać wartości z zakresu od 0 do 255.




Rys. 3.3 Przechowywana wartość może być wyświetlona, jako znak „c" lub liczba „67".

## 3.8. Przekształcenia typów

Bardzo często w programie następuje przekształcenie danej z pierwotnie zdefiniowanego typu do innego. Przekształcenia typów dzielimy na ***przekształcenia jawne*** i ***przekształcenia niejawne***. Przekształcenia typów mogą powodować, że ilość bajtów potrzebnych do zapisania tej danej (rozmiar danej) może się zwiększyć lub zmniejszyć. Operacje tego typu mogą być niebezpieczne, czasem niemożliwe, w wielu przypadkach kompilator wysyła ostrzeżenia o niebezpieczeństwie.

Najczęściej z przekształceniami typów mamy do czynienia podczas wykonywania operacji arytmetycznych. Rozważmy następujący przykład:

```
suma = x + y ;
```

Jeżeli zmienne **x** i **y** są tego samego typu, np. typu **int**, wartość **x + y** jest także typu **int**. W przypadku, gdy **x** zdefiniowano, jako typ **short**, a **y**, jako typ **int**, sumowanie **x + y** jest wyrażeniem mieszanym. W takim przypadku zmienna **x** jest przekształcana do typu **int** i wartość **x + y** jest typu **int.** Należy podkreślić, że przechowywana wartość **x** w pamięci nie jest zmieniana! Jest to tylko chwilowa zamiana typu, wykonywana na potrzeby żądanej operacji ( tworzona jest odpowiednia kopia zmiennej **x** o żądanym typie). W operacjach

arytmetycznych mamy do czynienia z przekształcaniem typów zgodnymi ze ściśle określonymi dla danego typu kompilatora regułami. Zmienna, której typ ma mniejszy rozmiar jest przekształcana do typu, który ma większy rozmiar (jest to nazywane ***promocją typu***).
Najczęściej hierarchia typów ma postać:

```
int < unsigned < long < unsigned long < float < double
```

Podstawowe reguły dotyczące promocji typów są następujące:

- każdy typ **char** lub **short** jest promowany do typu **int**
- każdy **unsigned char** lub **unsigned short** jest promowany do typu **unsigned**
- jeżeli mimo takich promocji, mamy wciąż wyrażenia mieszane, jest wykonywana konwersja, zgodna z podaną hierarchia typów.

Rozpatrzymy następujący przykład. W programie zadeklarowano następujące zmienne:

```
char a1;
unsigned a2;
int a3;
short a4;
long a5;
float a6;
double a7;
```

Utworzymy wyrażenia złożone z mieszanych typów i określimy typ po konwersji:

| Wyrażenie | typ |
|---|---|
| `a1 – a4 / a3` | `int` |
| `3 * a2 – a3` | `unsigned` |
| `3.0 * a2 – a3` | `double` |
| `3 * a6 – a3` | `float` |
| `a1 + 1` | `int` |
| `a1 + 1.0` | `double` |
| `3 * a4 * a5` | `long` |

Podczas wykonywania instrukcji z wyrażeniami zawierającymi zmienne mieszanych typów, wykonywane są niejawne konwersje. W instrukcji przypisania, takiej jak np.:

```
a7 = a3
```

typ **int** (zmienna **a3**) będzie przekształcony do typu **double** (zmienna **a7**). Tego typu konwersja przebiega bezproblemowo, ponieważ mniejszy rozmiar jest promowany do większego. Ale możliwa jest także degradacja (zmiana typu o rozmiarze większym do mniejszego).

W takim przypadku mogą wystąpić problemy. Zmienna **char** może przechowywać liczbę 113, ale nie 22222. Gdy typy zmiennoprzecinkowe są poddawane konwersji do typu całkowitego, tracona jest część ułamkowa, liczby są obcinane zawsze w kierunku zera. Liczby takie jak 13.13 i 13.99 są zamieniane na 13, a liczba -13.5 jest zamieniana na -13. Tego typu niuanse są ilustrowane programem pokazanym na listingu 3.5

Listing 3.5 Konwersja typów.

```
//rzutowanie typow
#include <conio.h>
#include <stdio.h>
int main()
{ char znak;
  int i;
  float liczba;
  liczba = i = znak = 'H';
  printf("znak = %c, i = %d, liczba = %4.2f\n",
          znak, i, liczba);
  znak += 1;
  i = liczba +2*znak;
  liczba = 2.0*znak + i;
  printf("znak = %c, i = %d, liczba = %4.2f\n",
          znak, i, liczba);
  znak = 5212205.17;
  printf ("teraz znak = %c\n", znak);
  getche();
  return 0;
}
```

Po uruchomieniu programu mamy następujący wydruk:

```
znak = H,      i = 72   liczba = 72.00
znak = I,      i = 218  liczba = 364.00
```

W linii:

```
liczba = i = znak = 'H';
```

Litera ‘H” zostaje zapisana w zmiennej **znak**, jako jedno-bajtowa wartość kodu ASCII. Litera ‘H’ jest przekształcona do typu **int** i otrzymuje wartość 72. Wartość 72 jest przekształcana do liczby zmiennoprzecinkowej typu **float** i otrzymuje wartość 72.00.
W linii:

```
printf("znak = %c,i = %d,liczba = %4.2f\n",znak,i,liczba);
```

jest polecenie wydruku tych wartości. W liniach:

```
i = liczba +2*znak;
liczba = 2.0*znak + i;
```

zmienna znakowa ‘H’ zostaje przekształcona na liczbę całkowitą (typ **int,** wartość 72) do tej liczby dodano 1, co w wyniku daje liczbę 73 jako wartość zmiennej **znak.** Zmienna **i** otrzymuje wartość 218 (w wyniku sumowania : 72 + 2*73). Zmienna **liczba** otrzymuje wartość 364.00 ( w wyniku sumowania 2*73 + 218).
Należy unikać automatycznych konwersji. Programista ma odpowiednie narzędzie, kontrolujące konwersje – jest to metoda zwana rzutowaniem jawnym. ***Operator rzutowania*** ma postać:

```
(typ) wyrażenie
```

Rozważmy przykład. Niech zmienna **calaLiczba** będzie typu **int.** Automatyczna (niejawna) konwersja ma postać:

```
calaLiczba = 1.13 + 13.13;
```

Liczby 1.13 i 13.13 są sumowane, co daje 14.16, ten wynik jest obcinany do 14 i przypisany do zmiennej **calaLiczba,** która jest typu **int.** Sumowanie z wykorzystaniem operatora rzutowania może mieć postać:

```
calaLiczba = (int) 1.13 + (int) 13.13;
```

za pomocą operatora rzutowania **(int)** liczby 1.13 i 13.13 są przekształcone do liczb całkowitych 1 i 13 a następnie dodane.

## 3.9. Stałe i zmienne łańcuchowe

***Łańcuchem*** nazywamy ciąg znaków, jest to forma danych używanych do przechowywania tekstów. W odróżnieniu od kompilatorów innego typu, w języku C nie ma odrębnego typu danych dla łańcuchów. W języku C łańcuch jest tablicą typu **char**. Ciąg znaków tworzących łańcuch musi być zawarty między znakami cudzysłowu:

```
”rezultat =”            ”Jan Kowalski”
```

Przypominamy, że stała znakowa (w przeciwieństwie do stałej łańcuchowej) jest ograniczona apostrofami!
Każdy łańcuch w sposób niejawny zakończony jest znakiem **null** (zapis **\0,** ten zapis traktowany jest jak jeden znak). Jeżeli chcemy operować łańcuchem musimy zadeklarować go, jako tablicę jednowymiarową. Przykład pokazuje sposób używania zmiennych łańcuchowych. Program czyta łańcuch, który wprowadzany jest z klawiatury i wypisuje go na ekranie.
W programie zmienna łańcuchowa zadeklarowana jest, jako tablica:

```
char malarz[60];
```

Tablica **malarz[60]** jest tablicą o długości 60 znaków, ale możemy wprowadzić maksymalnie 59 znaków (ostatni bajt zarezerwowany jest na znak **\0),** nieuwzględnienie tego faktu prowadzi często do przykrych pomyłek.

Listing 3.6 Operowanie łańcuchem znaków.

```
/* operowanie lancuchami */
#include <stdio.h>
void main(void)
{ char malarz[60];
  puts("\nPodaj nazwisko malarza :   ");
  gets(malarz);
  puts("Wielkim holenderskim malarzem byl ");
  puts(malarz);
  printf("\nWielkim holenderskim malarzem
          byl %s",malarz);
}
```

Do wypisania komunikatu zastosowano funkcję **puts()**:

```
puts("\nPodaj nazwisko malarza :   ");
puts("Wielkim holenderskim malarzem byl ");
```

Funkcja wyjścia **puts()** jest jedną z wielu funkcji bibliotecznych służących do obsługi łańcuchów. Może ona wypisywać tylko jeden łańcuch, nie ma możliwości formatowania łańcucha. Oczywiście do wyświetlania łańcucha możemy stosować także funkcje **printf().** Po wykonaniu zadania, funkcja **puts()** ustawia kursor w początku nowego wiersza (ta akcja zależy od typu kompilatora). Do wprowadzania łańcuch zastosowano funkcję wejścia **gets()**:

```
gets(malarz);
```

Ta funkcja (nie jest ona taka rozbudowana jak **scanf()**) wczytuje łańcuch. Wprowadzanie łańcucha jest zakończone, gdy zostanie naciśniety klawisz Enter (spacje i tabulator są akceptowane, jako składowe łańcucha). Aby pokazać różnice w działaniu funkcji **puts()** i **printf()** wprowadzono dodatkową instrukcję:

```
printf("\nWielkim holenderskim malarzem byl %s",malarz);
```

Znak **\n** powoduje, że napis pokaże się w nowym wierszu, specyfikator formatu jest:

```
%s – wyświetla łańcuch
```

Działanie programu jest następujące:

```
Podaj nazwisko malarza :
Hieronim Bosch
Wielkim holenderskim malarzem był
Hieronim Bosch
Wielkim holenderskim malarzem był Hieronim Bosch
```

## 3.10. Typ wyliczeniowy

Za pomocą słowa kluczowego ***enum***, tworzony jes***t typ wyliczeniowy.*** Dzięki temu mamy możliwość tworzenia skończonego zbioru i deklarowania zmiennych, które są elementami tego zbioru. Elementami tego zbioru są stałe całkowite. Różnica pomiędzy elementami wyliczeniowymi a stałymi zadeklarowanymi, jako **const** polega na tym, że dla elementu wyliczeniowego nie przydziela się adresowalnego obszaru pamięci. Zadaniem typów wyliczeniowych jest zwiększenie czytelności programu przez przyporządkowanie liczbom bardziej czytelnych nazw.
Typ wyliczeniowy definiowany jest następująco:

```
enum nazw_typu { lista_wyliczenia } lista_zmiennych
```

Deklaracja typu wyliczeniowego zaczyna się słowem kluczowym **enum z** następującą po nim nazwą typu, po nim następuje nawias klamrowy, wewnątrz nawiasu klamrowego umieszczone są elementy wyliczeniowe. Domyślną wartością pierwszego elementu jest zero. Każdy następny element na liście otrzymuje wartość większą o jeden od poprzedniego. Nazwa typu jak i lista zmiennych są opcjonalne. Przykładem jest zapis:

```
enum { falsz, prawda};
```

W tym przykładzie element wyliczeniowy **falsz** otrzymał wartość 0, a element **prawda** – wartość 1.
W deklaracji:

```
enum {f alsz, nic = 0, sukces, prawda =1}
```

Elementy **falsz** i **nic** otrzymują wartość 0 a elementy **sukces** i **prawda** – wartość 1.
Stosowanie typów wyliczeniowych zilustrujemy przykładem. Możemy dni tygodnia ponumerować od 0 do 6 i używać tych liczb w programie. Dzięki typowi wyliczeniowemu możemy liczby zastąpić nazwami. Program z listingu 3.7 wykorzystujący typ **enum** drukuje następny dzień tygodnia po wprowadzeniu bieżącego dnia.
Oto przykładowy wynik działania programu:

```
Dni tygodnia to : pon wto sro czw pia sob nie
Podaj dzien:
nie
Nastepny dzien to
poniedziałek
```

Na pierwszy rzut oka program jest udziwniony. Spowodowane jest to faktem, że w przypadku wykonywania operacji wejścia/wyjścia na zmiennych wyliczeniowych nie można określić wartości symboli wyliczeniowych. Wynikła potrzeba zastosowania w naszym programie specjalnych konstrukcji.

Listing 3.7 Typ wyliczeniowy (**enum**).

```
//Typ wyliczeniowy: enum
#include <conio.h>
#include <stdio.h>
#include <string.h>
int main()
{ enum dzien {pon, wto, sro, czw, pia,sob,nie};
  const char * dni[] = {"pon", "wto", "sro", "czw",
                        "pia", "sob", "nie"};
  int i,nast;
  char wybor[10];
  puts("Dni tygodnia to: pon wto sro czw pia sob
        nie \n");
  puts("Podaj dzien : ");
  gets(wybor);
  for (i=0;i<8;i++) {
    if (strcmp(wybor, dni[i]) ==0)  nast = i;
  }
  puts("Nastepny dzien to: ");
  switch(nast)
  {  case pon: puts("wtorek");
       break;
   case wto: puts("sroda");
       break;
   case sro: puts("czwartek");
       break;
   case czw: puts("piatek");
       break;
   case pia: puts("sobota");
       break;
   case sob: puts("niedziela");
       break;
   case nie: puts("poniedzialek");
       break;
  }
  getche();
  return 0;
}
```

W linii:

```
const char * dni[] = {"pon", "wto", "sro", "czw",
                      "pia", "sob", "nie"};
```

zdefiniowano tablicę **dni[ ],** ponieważ typy wyliczeniowe mogą być wykorzystywane jedynie wewnętrznie przez sam program. Zmienne wyliczeniowe są wygodnymi etykietami instrukcji **case.** Wartość stałej wyliczeniowej np. 0 jest także indeksem tablicy **dni[ ].**

Po wprowadzeniu z klawiatury nazwy dnia tygodnia:

```
char wybor[10];
puts("Dni tygodnia to: pon wto sro czw pia sob nie\n");
puts("Podaj dzien : ");
gets(wybor);
```

wykorzystana jest funkcja biblioteczna **strcmp()** pracująca w pętli **for** do wyznaczenia interesującej nas wartości. Ten fragment ma postać:

```
for (i=0;i<8;i++) {
  if (strcmp(wybor, dni[i]) ==0)  nast = i;
}
```

Ustalona zmienna **nast** spowoduje poprawne wykonanie instrukcji **switch.**

## 3.11. Wyrażenia

Na zdefiniowanych typach danych można wykonywać różnego typu podstawowe operacje. Do wykonywania operacji na danych służy zbiór operatorów zdefiniowanych pierwotnie i zbiór instrukcji. Operatory, stałe i zmienne są składnikami wyrażeń. W języku C wyrażenie jest dowolną kombinacją tych elementów. Programy przetwarzają informacje. W praktycznym podejściu program przekształca dane wejściowe tak, aby użytkownik otrzymał potrzebną informację w postaci danych wyjściowych (lub żądanej akcji). Musimy poinformować komputer jak ma działać na danych. Te działanie związane są z wyrażeniami. Wyrażenie jest elementem języka C i może przyjąć jedną z następujących postaci:

- nazwa zmiennej
- nazwa tablicy
- nazwa funkcji
- stała
- wywołanie funkcji
- odwołanie do elementu tablicy
- odwołanie do elementu struktury
- kombinacja stałych, zmiennych, operatorów i wywołań funkcji

Przykłady wyrażeń:

```
ax + bx
13*13 + 1313*ax
sin(ax) + cos(2*bx)
```

Każde wyrażenie posiada wartość. W przykładzie wyrażenie **ax + bx** ma wartość, która zależy od wartości **ax** i **bx**. Jeżeli np. napiszemy:

```
ax = 1;
bx = 2;
printf(”Suma=%d”, ax + bx);
```

to zostanie wydrukowany napis:

```
Suma = 3
```

Wartość wyrażenia jest obliczana przez wykonanie wszystkich działań zgodnie z priorytetami i kierunkami wiązania operatorów.
Wyrażenia relacyjne takie jak np.:

```
x >13
```

także mają wartość. Jeżeli powyższe wyrażenie jest prawdziwe to ma wartość 1, gdy jest fałszywe to ma wartość 0. W Tabeli 3.6 pokazano kilka przykładów wyrażeń i ich wartości.

Tabela 3.6 Wartości wyrażeń.

| Wyrażenie | Wartość |
|---|---|
| -10 + 5 | -5 |
| c = 3 + 8 | 11 |
| 13 > 3 | 1 |
| 6 + (3 + 8) | 17 |

## 3.12. Instrukcje

***Instrukcjami*** nazywamy te linie tekstu programu, które powodują jakąś akcję komputera. Instrukcje są najmniejszymi wykonywalnymi jednostkami programu w języku C. Praktycznie, wyrażenie zakończone średnikiem jest uważane za instrukcję. W zasadzie instrukcje wykonywane są w kolejności ich występowania w tekście programu. Najprostszą instrukcją jest instrukcja pusta. Ma ona następującą postać:

```
;   //przykład instrukcji pustej
```

Korzystamy z instrukcji pustych, gdy składnia języka wymaga instrukcji w programie, chociaż w konkretnej sytuacji jej nie potrzebujemy. Taka sytuacja występuje często w przypadku użycia pętli **while** oraz **for.** W takim przypadku główna część pętli jest pusta.
W przykładzie

```
for ( ; *str == ' ' ; str++) ;
```

pętla usuwa wszystkie początkowe spacje ze strumienia wskazywanego przez zmienną **str**. Główna część tej pętli jest pusta.
W programach czasami stosowane są pętle opóźniające, stosujemy je, gdy chcemy spowolnić wykonywanie programu w danym miejscu. Przykładem takiej pętli jest instrukcja:

```
for ( x = 0; x < wartość; x++) ;
```

Deklaracja zakończona średnikiem jest ***instrukcją deklaracji***, jedyną instrukcją, która może być określona na zewnątrz funkcji.

Wyrażenie zakończone średnikiem jest ***instrukcją wyrażenia.***
W języku C występują ***instrukcje proste***:

```
suma = zmienn_a + zmienna_b;
```

oraz ***złożone*** :

```
{
   k++ ;
   j++  ;
}
```

***Instrukcja złożona*** (zwana też ***blokiem***) zaznaczona jest parą nawiasów klamrowych, zawiera kilka instrukcji prostych. Instrukcję złożoną traktuje się tak, jak pojedynczą instrukcję. Ponieważ język C uznaje za instrukcję każde wyrażenie zakończone średnikiem, wynika, że następujące zapisy są poprawnymi instrukcjami:

```
13;
13 + 33;
```

Takie instrukcje nie są istotne w programie. Instrukcje najczęściej zmieniają wartości lub wywołują funkcje, jak to pokazano następnym (sensownym) przykładzie:

```
x = 66;
++x;
printf( " x =  %d ", x);
```

Każda instrukcja jest kompletnym poleceniem, nie wszystkie kompletne polecenia są instrukcjami. Ilustruje to następny przykład:

```
x = 13 + ( y = 5) ;
```

W tej instrukcji podwyrażenie ( y = 5 ) jest kompletnym polecenie, jest jednak tylko częścią pełnej instrukcji.

Rozróżnia się instrukcje takie jak:

- instrukcja przypisania
- instrukcja warunkowa (**if**)
- instrukcja iteracyjna (**while**, **do while**, **for** )
- instrukcje zaniechania (**break**)
- instrukcje kontynuowania (**continue**)
- instrukcja wyboru (**switch**)
- instrukcja wywołania funkcji
- instrukcja powrotu (**return**)
- instrukcja pusta ( **;** )
- instrukcja skoku ( **goto**)

## 3.13. Operatory

W wyrażeniach określone są operacje, tj. rodzaje i kolejność obliczeń. Takie działania (operacje) wykonywane są na argumentach (zwanych też ***operandami***). Konkretne działanie reprezentuje operator, który działa na argument.
Np. operator arytmetyczny ”+” powoduje dodanie dwóch liczb, operator odejmowania ”–” powoduje odjęcie dwóch liczb. Operatory mają różne kolejności wykonywania – priorytety. W podręcznikach najczęściej wymienia się operatory w zależności od priorytetu: od największego do najmniejszego.
Operator, który działa na jeden argument nazywamy ***operatorem jednoargumentowym*** (***unarnym***). Operator działający na dwa argumenty nazywamy ***operatorem dwuargumentowym*** (albo ***binarnym***), w tym przypadku rozróżniamy argumenty i klasyfikujemy je, jako ***operand lewy*** i ***operand prawy***. Związane to jest z zasadą łączności operatorów. Wyróżniamy operatory ***łączone prawostronnie*** i operatory ***łączone lewostronnie***. W języku C występuje duża ilość operatorów (w standardzie mamy 45 operatorów, zgrupowanych w kilkunastu grupach). W tabeli 3.7 pokazane są priorytety operatorów. Operatory są podzielone na 16 kategorii. Kategoria pierwsza ma najwyższy priorytet, kategoria druga ma niższy niż kategoria pierwsza priorytet i tak dalej. Operatory należące do tej samej kategorii mają taki sam priorytet. Operatory unarne (grupa 2), warunku (grupa 14) oraz przypisania (grupa 15) są prawostronnie łączone, pozostałe są łączone lewostronnie.

Tabela 3.7 Priorytety operatorów.

| Kategoria/typ | Operator | Działanie/Znaczenie | Przykład |
|---|---|---|---|
| 1. Najwyższa | ( ) | wywołanie funkcji | getc(znak) |
| | [ ] | odwołanie do elementu tablicy | tab[13] |
| | - > | wskaźnik do elementu struktury | x_wsk->y |
| | :: | operator zakresu (C++ ) | X::fun |
| | . | odwołanie do elementu struktury | x_w.y |
| 2. Unarny | ! | logiczna negacja (NOT) | !mam |
| | ~ | uzupełnienie do jeden | ~0xff |
| | + | unarny plus | +k |
| | - | unarny minus | -k |
| | ++ | inkrementacja | k++ |
| | -- | dekrementacja | k-- |
| | & | adres elementu | &s |
| | * | wskazanie pośrednie | *p_wsk |
| | sizeof | rozmiar | sizeof(k) |
| | new | (dynamiczna pamięć) | |
| | delete | (dynamiczna pamięć) | |

| | | | |
|---|---|---|---|
| 3. Operatory dostępu | * | C++ dereferencja | |
| | ->* | C++ dereferencja | |
| 4. Arytmetyczny | * | mnożenie | x * y |
| | / | dzielenie | k / n |
| | % | operacja modulo | k % n |
| 5. Arytmetyczny | + | dodawanie | k + n |
| | - | odejmowanie | k - n |
| 6. Przesunięcie | << | przesunięcie bitowe w lewo | k <<2 |
| | >> | przesunięcie bitowe w prawo | k >>2 |
| 7. Relacje | < | mniejszy niż | k < x |
| | <= | mniejszy niż lub równy | k <=13 |
| | > | większy niż | k > 13 |
| | >= | większy niż lub równy | k >=w |
| 8. Równości | == | równy | if (k ==13) |
| | != | nierówny | if (k != 13) |
| 9. | & | bitowy AND | zx & 033 |
| 10. | ^ | bitowy XOR | zx ^ 0317 |
| 11. | \| | bitowy OR | zx \| 0333 |
| 12. | && | logiczny AND | k==13 && m==77 |
| 13. | \|\| | logiczny OR | k==13 \|\| m==77 |
| 14. Warunkowy | ?: | wyrażenie warunkowe | j >13 ? j : k |
| 15. Przypisania | = | przypisanie | zx = 13 |
| | *= | mnożenie, potem przypisanie | k *= 13 |
| | /= | dzielenie, potem przypisanie | k /= 13 |
| | %= | modulo, potem przypisanie | k %= 4 |
| | += | dodawanie, potem przypisanie | k += 13 |
| | -= | odejmowanie, potem przypisanie | k -= 13 |
| | &= | iloczyn bitowy, potem przypisanie | k &= 0333 |
| | ^= | suma mod 2 , potem przypisanie | k ^= 0317 |
| | \|= | iloczyn logiczny, potem przypisanie | k \|= 0177 |
| | <<= | przesuń w lewo, potem przypisanie | k <<= 2 |
| | >>= | przesuń w prawo, potem przypisanie | k >>= 2 |
| 16. Przecinkowy | , | oblicz i odrzuć | k = 13 , zx |

**Operatory arytmetyczne**

Do dyspozycji mamy pięć operatorów arytmetycznych. W języku C nie występuje operator potęgowania, standardowa biblioteka C zawiera funkcję **pow(),** dzięki której można wykonać potęgowanie. Oczywiście dzielenie przez zero jest zabronione.

Tabela 3.8. Operatory arytmetyczne

| Grupa | Symbol | Działanie | Łączność | Przykład |
|---|---|---|---|---|
| 4. | * | mnożenie | lewostronna | x * y |
| 4. | / | dzielenie | lewostronna | k / n |
| 4. | % | operacja modulo | lewostronna | k % n |
| 5. | + | dodawanie | lewostronna | k + n |
| 5. | - | odejmowanie | lewostronna | k – n |

Ten zestaw operatorów służy głównie do wykonywania działań arytmetycznych, tak jak to jest opisywane w podręcznikach matematyki. Operatory arytmetyczne zaliczane są do grupy operatorów binarnych, ponieważ wymagają dwóch operandów (argumentów). Aby wykonać działanie arytmetyczne takie jak np. odejmowanie potrzebujemy odjemnej i odjemnika:

```
a – b
```

gdzie **a** i **b** są operandami. Operatory dzielenia : ” **/** ” i ” **%**” mają specjalne, ściśle określone działania. Operator dzielenia całkowitego ” / ” zwraca wartość całkowitą, po odrzuceniu części ułamkowej:

wartością wyrażenia 1 / 2 jest wartość 0
wartością wyrażenia 13 / 11 jest wartość 1
wartością wyrażenia -5 / 2 jest wartość -2

Operator **%** zwany ***operatorem modulo*** pozwala obliczyć resztę z dzielenia. Wyrażenie:

a % b czytamy jako „a modulo b”

Ściśle mówiąc, jeżeli **a** i **b** są liczbami dodatnimi, całkowitymi to w wyniku działania tego operatora otrzymamy resztę z dzielenia **a** przez **b:**

wartością wyrażenia 11 % 5 jest 1
wartością wyrażenia 10 % 5 jest 0

Dla przypomnienia, w naszym przykładzie dzielimy 11 przez 5 , otrzymujemy 2 oraz resztę 1.

Dla dodatnich liczb całkowitych operacje modulo definiujemy następująco:

*wartością wyrażenia ((a / b) * b) + ( a % b) jest wartość a.*

Jeżeli **a** lub **b** są ujemne, to wynik jest zdefiniowany, ale zależy od systemu komputerowego. Tak jak to obowiązuje w arytmetyce, nie można dzielić przez zero.

Klasyczne zastosowanie operatora dzielenia modulo ilustrowane jest rozwiązaniem zagadnienia:, w jaki sposób wydać resztę, mając w kasie określony zbiór banknotów.

Listing 3.8 Dzielenie z resztą.

```
/* program rozmienia wyplate na banknoty*/
#include <stdio.h>
#include <conio.h>
void main(void)
{ int kwota;              //kwota do rozmienienia
  int b20, b10, b5, b1;  //ilosc banknotow 20,10,
                         //5 i 1 zlotowych
  int reszta20, reszta10;
  printf("Kwota do wydania musi byc liczba calkowita
          np. 66");
  printf("\npodaj kwote do wydania : ");
  scanf("%d", &kwota);
  b20 = kwota/20;
  reszta20 = kwota % 20;
  b10 = reszta20/10;
  reszta10 = reszta20 % 10;
  b5 = reszta10/5;
  b1 = reszta10 % 5;
  printf("Kwota do wydania %d zl :\n",kwota);
  printf("Wydaj dwudziestki, ilosc =  %d \n",b20);
  printf("Wydaj dziesiatki,  ilosc =  %d \n",b10);
  printf("Wydaj piatki ,     ilosc =  %d \n",b5);
  printf("Wydaj zlotowki ,   ilosc =  %d \n",b1);
  getch();
}
```

Po wprowadzeniu z klawiatury kwoty, np. 97 otrzymamy wydruk:

```
Kwota do wydania musi być liczba calkowita np. 66
Podaj kwote do wydania :  97
Kwota do wydania  97 zl
Wydaj dwudziestki , ilosc = 4
Wydaj dziesiatki ,  ilosc = 1
Wydaj piatki ,      ilosc = 1
Wydaj zlotowki ,    ilosc = 2
```

Działanie programu jest proste.

Można sprawdzić ręcznie następujący fragment, dla zmiennej kwota = 97:

```
b20 = kwota/20;
reszta20 = kwota % 20;
b10 = reszta20/10;
reszta10 = reszta20 % 10;
b5 = reszta10/5;
b1 = reszta10 % 5;
```

Wykonamy „ręczną symulację" obliczeń:
instrukcja b20 = kwota/20; wynik: 97 / 20 = 4 (reszta 17)
instrukcja reszta20 = kwota % 20; wynik: 97 % 20 = 17 (dzielenie modulo)
instrukcja b10 = reszta2 0/10; wynik: 17 / 10 = 1 (reszta 7)
instrukcja reszta10 = reszta20 % 10; wynik: 17 % 10 = 7 (dzielenie modulo)
instrukcja b5 = reszta10/5; wynik: 7 / 5 = 1 (reszta 2)
instrukcja b1 = reszta10 % 5; wynik: 7 % 5 = 2 (dzielenie modulo)
Ostatecznie mamy:

```
b20 =  4
b10 =  1
b5  =  1
b1  =  2
```

Wyrażenie może zawierać wiele operatorów. W takich sytuacjach podstawowe znaczenie ma pytanie jak działania są wykonywane. W wyrażeniu:

```
1 + 2 * 3
```

wynikiem jest liczba 7 (a nie 9 jak mógłby ktoś sądzić przyjmując, że działania wykonywane są kolejno do strony lewej do prawej). W języku C kolejność wykonywania operacji jest precyzyjnie ustalona – oznacza to, że pewne działania są wykonywane przed innymi, mimo, że w wyrażeniu pojawiają się później. Kolejność wykonywania działań jest podana w tabeli operatorów. W przypadkach wątpliwych rekomenduje się stosowanie nawiasów okrągłych. Program pokazany na listingu 3.9 pozwoli na sprawdzenie kolejności wykonywania działań. W wyniku wykonania tego programu mamy wydruk:

```
Testowanie kolejności działań

     wynik1 = -1
     wynik2 = 0
     wynik3 = 1
     wynik4 = 0
     wynik5 = 0
     wynik6 = 57
     wynik7 = 0
     wynik8 = 2
     wynik9 = 15
     wynik10 = -3
     wynik11 = -33
```

Listing 3.9 Kolejność wykonywanych operacji arytmetycznych.

```
#include <stdio.h>
int main()
{ int a = 3;    int b = -3;
  int c = 7;    int d = -19;
  int wynik;
  printf("\ntestowanie kolejnosci dzialan");
  wynik = a / b;
  printf("\nwynik1 = %d",wynik);
  wynik = c / b / a;
  printf("\nwynik2 = %d",wynik);
  wynik = c % a ;
  printf("\nwynik3 = %d",wynik);
  wynik = a % b;
  printf("\nwynik4 = %d",wynik);
  wynik = d / b % a;
  printf("\nwynik5 = %d",wynik);
  wynik = - a * d;
  printf("\nwynik6 = %d",wynik);
  wynik = a % - b * c;
  printf("\nwynik7 = %d",wynik);
  wynik = 9/c + -20/d;
  printf("\nwynik8 = %d",wynik);
  wynik = (-d%c-b/a*5+5);
  printf("\nwynik9 = %d",wynik);
  wynik = (- - - a);
  printf("\nwynik10 = %d",wynik);
  wynik = (a=b=c=-33);
  printf("\nwynik11 = %d",wynik);
  return 0;
}
```

**Arytmetyczne operatory przypisania oraz inkrementacji**

W teorii algorytmów rozpatruje się niezwykle użyteczne konstrukcje takie jak ***licznik*** i ***akumulator***.

***Licznik*** to zmienna, której nadano określoną wartość początkową i która zwiększa swoją wartość o 1 po każdorazowym zajściu jakiegoś zdarzenia.

Zapis licznika jest następujący:

```
zmienna  =  zmienna + 1
```

Ponieważ w tym zapisie mamy operator przypisania = , w programie najpierw wyliczana jest prawa strona :

```
zmienna + 1
```

a następnie wyliczana wartość jest przypisana zmiennej znajdującej się po lewej stronie operatora przypisania.

Po wykonaniu tej instrukcji z licznikiem, nowa wartość zmiennej jest równa starej wartości zwiększonej o jeden. Oczywiście liczniki mogą zmniejszać swoja wartość, wobec tego mamy analogiczną konstrukcję:

```
zmienna = zmienna - 1
```

***Akumulator*** to zmienna mająca określoną wartość początkową i zwiększająca (lub zmniejszająca) swą wartość o konkretną inną wartość po każdym zajściu określonego zdarzenia. Postać akumulatora jest następująca:

```
zmienna1 = zmienna1 + zmienna2
```

lub

```
zmienna1 = zmienna1 – zmienna2
```

Nazwa akumulator związana jest z faktem, że dodawane wartości kumulują się.

Listing 3.10 Licznik i akumulator.

```
/* licznik i akumulator */
#include <stdio.h>
int main()
{ int licznik=0;
  float liczba,suma,srednia;
  suma = 0.0;
  printf("\npodaj liczbe : ");  scanf("%f",&liczba);
  suma = suma + liczba;
  licznik = licznik + 1;
  printf("\npodaj liczbe : ");  scanf("%f",&liczba);
  suma = suma + liczba;
  licznik = licznik + 1;
  printf("\npodaj liczbe : ");  scanf("%f",&liczba);
  suma = suma + liczba;
  licznik = licznik + 1;
  srednia = suma / licznik;
  printf("\nsrednia = %f dla %dliczb",
          srednia,licznik);
  return 0;
}
```

Wykorzystanie akumulatora ilustruje program z listingu 3.10, obliczamy średnią z trzech liczb. Po uruchomieniu tego programu otrzymujemy wydruk :

```
podaj liczbe : 50
podaj liczbe : 100
podaj liczbe : 50
srednia = 66.666664 dla 3 liczb
```

Ponieważ w programach korzysta się bardzo często z liczników i akumulatorów, w języku C udostępniono zestaw operatorów, ułatwiający pisanie zwartych instrukcji z tymi konstrukcjami.

***Operatory zwiększania*** ++ i ***zmniejszania*** **--** są operatorami unarnymi. Te operatory stosować można do zmiennych, ale nie wolno używać ich do stałych i wyrażeń. Przykłady użycia tych operatorów:

```
++k
--k
k++
k--
```

Instrukcja **i = i + 1**; jest równoważna instrukcji **++ i**;
Jest istotna różnica pomiędzy zapisem **++a** oraz **a++**. Mówimy o formie ***przedrostkowej*** lub o ***przyrostkowej***.

*Operacja* ***++a*** *zwiększa wartość zmiennej* ***a*** *o 1 przed jej użyciem.*
*Operacja* ***a++*** *zwiększa wartość zmiennej* ***a*** *o 1 po użyciu jej wartości.*

Te same zasady stosują się do operatora zmniejszania. Jak zwięzły może być zapis, gdy stosujemy omawiane operatory pokazuje przykład.
Zapis:

```
suma = ++ licznik;
```

jest równoważny dwóm instrukcjom :

```
licznik = licznik + 1;
suma = licznik;
```

Należy zdawać sobie sprawę, że wyrażenie takie jak np. **++a** zmienia swoją wartość w pamięci. Program pokazuje wybrane aspekty stosowania operatorów **++** i **--** .

Listing 3.11 Operatory inkrementacji.

```
/* operatory ++ i -- */
#include <stdio.h>
int main()
{ int a, b, c;
  a = b = c = 0;
  printf("\ntest operatorow ++ i -- ");
  a = ++b + ++c;
  printf("\n %d %d %d",a,b,c);
  a = b++ + c++;
  printf("\n %d %d %d",a,b,c);
  a = ++b + c++;
  printf("\n %d %d %d",a,b,c);
  a = b-- + --c;
  printf("\n %d %d %d",a,b,c);
  a = ++c + c;
  printf("\n %d %d %d",a,b,c);
  return 0;
}
```

Otrzymany wydruk z pokazanego programu ma postać:

```
test operatorow ++  i  --
2   1   1
2   2   2
5   3   3
5   2   2
6   2   3
```

Ostatni wynik (6 2 3) zależny jest od systemu. Jeżeli rozważymy zapis:

```
x = ++y + y;
```

to z zapisu wynika, że **y** powinno być zwiększone przed użyciem. Ale **y** występuje dwa razy w wyrażeniu. Obliczenie **x** zależy od systemu. Generalnie taka instrukcja jest przykładem nieeleganckiego programowania, należy raczej zastosować zapis:

```
++y;
x = y + y;
```

lub

```
x = 2 * (++y);
```

Dużym udogodnieniem w języku C są ***operatory przypisania***. Operatory przypisania gwarantują krótszy zapis a w wielu przypadkach instrukcja wykonywana jest szybciej. W instrukcji:

```
suma = suma + dana1:
```

wartość zmiennej **suma** jest dodawana do wartości zmiennej **dana1**, a wynik dodawania jest zapisywany w zmiennej **suma**. W języku C pokazana instrukcja może być napisana w następujący sposób:

```
suma += dana1
```

Operator ”+=” jest arytmetycznym operatorem przypisania. Wszystkie operatory arytmetyczne mogą być łączone z operatorem przypisania ”=”. Mamy 5 arytmetycznych operatorów przypisania, pokazanych w tabeli 3.9

Tabela 3.9 Arytmetyczne operatory przypisania.

| Grupa | symbol | opis | przykład | znaczenie |
|---|---|---|---|---|
| 15 | += | przypisanie sumy | a += b | a= a + b |
| 15 | -= | przypisanie różnicy | a -= b | a = a - b |
| 15 | ***=** | przypisanie iloczynu | a *= b | a = a * b |
| 15 | /= | przypisanie ilorazu | a /= b | a = a / b |
| 15 | %= | przypisanie reszty | a %= b | a = a % b |

Oprócz arytmetycznych operatorów przypisania w języku C występują inne jeszcze operatory przypisania.

Patrząc formalnie na operatory przypisania, mamy do czynienia z konstrukcją:

```
zmienna   operator=  wyrażenie
```

która jest równoważna zapisowi :

```
zmienna = zmienna operator (wyrażenie)
```

Jeżeli mamy wyrażenie :

```
y *= 13 + x
```

to równoważny zapis ma postać :

```
y = y*(13 + x)
```

czyli

$y = (y*13 + y*x)$

Listing 3.12 Arytmetyczne operatory przypisania.

```
/* operatory przypisania */
#include <stdio.h>
#include <conio.h>

int main()
{ int a,b,c;
  a=2;   b=4;   c=6;
  printf("\nArytmetyczne Operatory Przypisania\n");
  printf("\nstartowe c = %i  startowe a = %i ",c,a);
  c += a;
  printf("\ndla  (c+=a) mamy  c = %i",c);
  printf("\nstartowe c = %i  startowe a = %i ",c,a);
  c -= a;
  printf("\ndla  (c-=a) mamy  c = %i",c);
  printf("\nstartowe c = %i  startowe a = %i ",c,a);
  c *= a;
  printf("\ndla  (c*=a) mamy  c = %i",c);
  printf("\nstartowe c = %i  startowe a = %i ",c,a);
  c /= a;
  printf("\ndla  (c/=a) mamy  c = %i",c);
  printf("\nstartowe c = %i  startowe b = %i ",c,b);
  c %= b;
  printf("\ndla  (c%%=b) mamy  c = %i ",c);
  printf("\nstartowe c = %i startowe a = %i ",c,a);
  c *= ++a + 3;
  printf("\ndla (c*=++a+3) mamy c = %i ",c);
  getch();
  return 0;
}
```

Należy unikać instrukcji takich jak:

```
a[++m] += 13;
```

gdyż nie jest ona w ogólności równoważna zapisowi:

```
a[++m] = a[++m] + 13;
```

ponieważ zależy ona od systemu, nie mówiąc, że jest to kodowanie mało eleganckie. Program pokazany na listingu 3.12 ilustruje wykorzystanie operatorów przypisania. Wynik działania tego programu jest następujący:

```
Arytmetyczne Operatory Przypisania
startowe c = 6    startowe a = 2
dla (c+=a) mamy c = 8
startowe c = 8    startowe a = 2
dla (c-=a) mamy c = 6
startowe c = 6    startowe a = 2
dla (c*=a) mamy c = 12
startowe c = 12    startowe a = 2
dla (c/=a) mamy c = 6
startowe c = 6    startowe b = 4
dla (c%=b) mamy c = 2
startowe c = 2    startowe a = 2
dla (c*=++a+3) mamy c = 12
```

**Operatory porównania**

Operatory porównania i relacji służą do obliczania wartości „prawda” lub „fałsz”. Warunek prawdziwy daje w wyniku liczbę 1, warunek fałszywy daje liczbę 0. W języku C występuje sześć operatorów porównania. Operatory porównania i relacji są podstawowymi elementami wyrażeń logicznych oraz wchodzą w skład wielu konstrukcji takich jak np. instrukcje **if** czy **while.** Lista operatorów porównania i relacji przedstawiona jest w tabeli 3.10.

Tabela 3.10 Operatory porównania

| Grupa | operator | opis | przykład | łączność |
|---|---|---|---|---|
| 8 | == | równe | a == b | lewostronna |
| 8 | != | różne | a != b | lewostronna |
| 7 | < | mniejsze | a < b | lewostronna |
| 7 | <= | mniejsze lub równe | a <= b | lewostronna |
| 7 | > | większe | a > b | lewostronna |
| 7 | >= | większe lub równe | a >= b | lewostronna |

**Operatory logiczne**

Wyrażenia warunkowe występujące w konstrukcjach **if** i **while** są najczęściej wyrażeniami relacyjnymi. W wielu sytuacjach celowe jest połączenie dwóch lub

więcej wyrażeń relacyjnych w jeden warunek. W wykonanie tego zadania pomagają ***operatory logiczne.*** W języku C istnieją trzy operatory logiczne.

Tabela 3.11 Operatory logiczne

| Grupa | operator | opis | przykład | łączność |
|---|---|---|---|---|
| 12 | && | logiczne AND | a && b | lewostronna |
| 13 | \| | logiczne OR | a \|\| b | lewostronna |
| 2 | ! | logiczne NOT | !a | prawostronna |

**Operatory bitowe**

Operatory bitowe pozwalają na bezpośrednie manipulowanie bitami. Pozwalają przetwarzać jedynie dane należące do typów całkowitych (włącznie z typem **char**). Operatory bitowe dzielimy na bitowe operatory logiczne i bitowe operatory przesunięcia. Operator bitowy działa na argumencie (argumentach) tak jak na uporządkowanym zbiorze bitów. W pamięci komputera każdy bit ma wartość albo 1, albo 0. Język C posiada zestaw sześciu operatorów bitowych.

Tabela 3.11. Operatory bitowe.

| Grupa | operator | opis | przykład | łączność |
|---|---|---|---|---|
| 9 | & | bitowa koniunkcja (AND) | a & b | lewostronna |
| 10 | ^ | bitowa różnica symetryczna | a ^ b | lewostronna |
| 11 | \| | bitowa alternatywa (OR) | a \| b | lewostronna |
| 2 | ~ | uzupełnienie jedynkowe | ~ a | prawostronna |
| 6 | << | przesuniecie w lewo | a << 4 | lewostronna |
| 6 | >> | przesunięcie w prawo | a >> 2 | prawostronna |

**Operatory przypisania bitowe**

Podobnie jak dla operatorów arytmetycznych, w języku C istnieją bitowe operatory przypisani. Tabela 3.12 pokazuje zestaw pięciu takich operatorów.

Tabela 3.12 Operatory bitowe przypisania.

| Grupa | operator | opis | przykład | znaczenie |
|---|---|---|---|---|
| 15 | <<= | przypisanie przesunięcia w lewo | a <<= b | a = a << b |
| 15 | >>= | przypisanie przesunięcia w prawo | a >>= b | a = a >> b |
| 15 | &= | przypisanie koniunkcji | a &= b | a = a & b |
| 15 | ^= | przypisanie różnicy symetryczna | a ^= b | a = a ^ b |
| 15 | \|= | przypisanie alternatywy | a \|= b | a = a \| b |

**Operatory rozmiaru**

Operator **sizeof** zwraca rozmiar operandu w bajtach. W języku C bajt zdefiniowany jest, jako długość typu **char.** W przeszłości ustalono, że jeden bajt jest równy ośmiu bitom, ale w aktualnie budowanych komputerach typ **char** może potrzebować więcej bitów. Dla naszych celów zakładamy tradycyjnie, że jeden bajt ma osiem bitów.
Operand może być konkretnym obiektem (wyrażenie, zmienna) lub nazwą typu (specyfikatorem typu). Poprawna konstrukcja może mieć postać:

```
sizeof (specyfikator_typu)
sizeof (wyrażenie)
```

Język C gwarantuje, że zachodzą następujące relacje:

```
sizeof(char) = 1
sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
sizeof(unsigned) = sizeof(int)
sizeof(float) ≤ sizeof(double)
```

**Operator konwersji**

Operator konwersji, często nazywany także operatorem rzutowania, pozwala na jawne dokonanie konwersji typów danych. Typowa konstrukcja z użyciem operatora konwersji ma postać:

```
(nazwa typu) wyrażenie
nazwa typu( wyrażenie)
```

**Operator warunkowy**

Język C pozwala na skrótowy zapis instrukcji **if....else** korzystając z ***operatora warunkowego*** **?:** . Jest to jedyny w języku C operator trójargumentowy. Operator składa się z dwóch symboli i wymaga trzech operandów. Operator warunkowy nosi też nazwę ***operatora arytmetycznego if***. Konstrukcja operatora warunkowego ma postać:

```
wyrażenie_warunkowe ? wyrażenie_na_tak :
wyrażenie_na_nie
```

Zapis ten ma następujące znaczenie:, jeśli **wyrażenie_warunkowe** jest prawdziwe (niezerowe) to całe wyrażenie warunkowe ma wartość taką jak **wyrażenie_na_tak**. Jeżeli **wyrażenie_warunkowe** jest fałszywe (równe zeru), całe wyrażenie warunkowe otrzymuje wartość **wyrażenia_na_nie.**
Poniższy przykład pozwala na wyznaczenie wartości bezwzględnej:

```
a = (b < 0) ? −b : b;
```

Powyższą instrukcję możemy rozumieć następująco:, jeśli **b** jest mniejsze od zera, niech **a** jest równe **–b**, w przeciwnym wypadku, niech **a = b**.

**Operator przecinkowy**

Wyrażenie przecinkowe jest ciągiem wyrażeń rozdzielonych przecinkami. Para wyrażeń oddzielonych operatorem przecinkowym obliczana jest od lewej strony do prawej. Wynikiem wyrażenia przecinkowego jest wartość ostatniego na prawo wyrażenia.

```
wyrażenie, wyrażenie
```

**Operator wskazywania**

Do grupy operatorów wskazywania w języku C należą cztery operatory pokazane w tabeli 3.13

Dostęp do składowych struktur jest możliwy dzięki ***operatorom dostępu do składowych.*** Dostęp realizujemy za pomocą ***operatora kropki*** (**.**) oraz ***operatora strzałki*** (->). Operator strzałki umożliwia dostęp do składowych struktury przez wskaźnik do obiektu.

Tabela 3.13. Operatory wskazywania.

| Grupa | operator | opis |
|---|---|---|
| 2 | & | operator adresu |
| 2 | * | operator adresowania pośredniego |
| 1 | . | operator składowej |
| 1 | -> | operator wskaźnikowy składowej |

# ROZDZIAŁ 4

## FORMATOWANE WEJŚCIE/ WYJŚCIE

## 4.1. Wstęp

Operacje wejścia/wyjścia w języku C nie wchodzą w skład języka. Programista może realizować operacje wejścia/wyjścia za pomocą odpowiednich funkcji i makrodefinicji zawartych w plikach bibliotecznych. W tym rozdziale koncentrować będziemy uwagę na operacjach konsolowych – zajmiemy się obsługą wprowadzania informacji z klawiatury oraz wyświetlaniem informacji na ekranie monitora. Wszystkie funkcje wejścia/wyjścia w zasadzie umieszczone są w pliku nagłówkowym stdio.h (ang. *standard input/output*), dlatego w każdym prawie programie napisanym w języku C mamy klasyczna dyrektywę preprocesora:

```
#include <studio.h>
```

Chociaż w początkowej fazie tworzenia języka C funkcje wejścia/wyjścia nie były zdefiniowane, a tworzenie tych funkcji pozostawiono producentom kompilatorów, to obecnie ustaliły się zbiory funkcji obsługujących operacje wejścia/wyjścia, które możemy praktycznie traktować jak elementy standardowe.

## 4.2. Formatowane wejście/ wyjście

W wielu zadaniach wykonywanych przez programy wymaga się, aby dane były wprowadzane do programu i wypisywane na ekranie monitora, zapisywane do plików na dyskach i innych trwałych nośnikach informacji bądź drukowane przez drukarkę. Programista ma dostęp do wyspecjalizowanych funkcji i makrodefinicji umożliwiających komunikowanie się z otoczeniem poprzez wykorzystanie ***bibliotek***. Odpowiednie biblioteki dostarczane są z kompilatorem, programista ma też możliwość korzystania z zewnętrznych bibliotek, które sam napisze albo zakupi w niezależnych firmach komputerowych. Obecnie omówimy najbardziej popularne funkcje i operacje wejścia i wyjścia, które są potrzebne do odwołań do najpopularniejszych urządzeń obecnych w każdym komputerze.

## 4.3. Proste operacje wejścia/wyjścia.

Podstawowym pojęciem w opisie operacji wejście/wyjście jest plik. Zawężając to pojęcie przyjmujemy następujące określenie:

***plik*** jest sekwencją bajtów danych
umieszczonych na dowolnym nośniku

To pojęcie nie odnosi się do czytania i pisania do portów, czyli adresów urządzeń peryferyjnych dołączonych do mikroprocesora.
Plik przyjmuje i udostępnia ciąg bajtów a zapis fizyczny nie wpływa na jego działanie.

Programując komputery typu IBM PC musimy pamiętać, że mamy dwa typy plików:

- **pliki tekstowe**
- **pliki binarne**

Konieczność dysponowania plikami tekstowymi i binarnymi powstała z powodu istnienia niekompatybilności pomiędzy systemem UNIX (język C był opracowany dla tego systemu) a systemem MS DOS i systemem CP/M (stary, praktycznie nieistniejący już system). Unix korzysta z własnej konwencji plików a powstałe później na potrzeby PC-tów systemy CP/M i MS DOS korzystają z własnych konwencji. Aby przenieść kompilator C na platformę MS DOS, firma BORLAND, aby usunąć tą niekompatybilność, zaproponowała rozwiązanie w postaci dwóch typów plików. Dla programów w języku C, pliki w trybie tekstowym wyglądają tak jak pliki w systemach UNIX. Pliki w trybie binarnym wyglądają tak jak pliki w systemach MS DOS. W plikach tekstowych każdy bajt jest interpretowany, jako znak ASCII, koniec pliku jest sygnalizowany za pomocą znaku **CTRL+Z**. W plikach binarnych, poszczególne bajty nie są interpretowane w żaden sposób. Aby odczytać zawartość pliku binarnego musimy wiedzieć, w jaki sposób dokonano zapisu. Pliki binarne ze względu na rozmiar są preferowane do zapisu danych numerycznych. Do zapisu np. liczby 30 000 w pliku tekstowym potrzebujemy 5 bajtów (ta liczba reprezentowana jest przez 5 znaków, na jeden znak potrzebujemy 1 bajt). W trybie binarnym potrzebujemy do zapisu jedynie 2 bajtów. W tym przypadku rozmiar pliku tekstowego jest 2.5 razy większy niż plik binarny. Plik binarny nie może być odczytany przez edytor tekstu. Dla programisty istotne są dwa zagadnienia różnicujące pliki tekstowe i binarne:

- przejście do nowej linii
- koniec pliku

W języku C znak nowej linii **\n** oznacza jednocześnie koniec linii. W pliku tekstowym w systemie MS DOS koniec linii określa para znaków:

- **CR** - powrót karetki
- **LF** - wysunięcie linii

Znaki **CR** i **LF** są reprezentowane w języku C, jako **\r** i **\n**. W popularnych kompilatorach rozwiązano ten problem dzięki odpowiednim procedurom obsługującym pliki - wykonywana jest translacja danych zapisanych w trybie tekstowym. Znak **CTRL+Z** oznacza koniec pliku otwartego w trybie tekstowym. Podczas czytania para znaków **CR** i **LF** jest zamieniana na znak **LF** (oznacza to znak nowej linii w języku C).

Przy zapisie, pojedynczy znak **LF** powoduje wysłanie pary znaków **CR** i **LF,** co umożliwia zachowanie poprawności pliku w systemie MS DOS. Tak, więc, programista w większości przypadków nie musi troszczyć się zbytnio o obsługę różnych typów plików. Jeżeli chcemy zapewnić przenoszenie oprogramowania musimy dokładnie znać opis każdej funkcji, ponieważ dostępne procedury dla funkcji takich jak np. **printf()** i **scanf()** mogą być dostępne jedynie w konkretnej implementacji kompilatora.

Dla programisty posługującego się kompilatorem języka C najczęściej dostępne są trzy procedury obsługujące operacje wejścia/wyjścia, zaimplementowane w bibliotekach:

- procedury strumieniowe
- procedury plikowe niskiego poziomu
- procedury obsługujące konsolę i porty wejście/wyjście

W tej części omówimy wybrane procedury strumieniowe. W procedurach strumieniowych plik traktuje się jak ciąg bajtów, najczęściej wykonywanie operacji wejścia/wyjścia wymaga ***bufora***. Procedury niskiego poziomu nie korzystają z buforów.

***Bufor*** jest obszarem w pamięci, tam zapisywane są dane z pliku, odczytywanie danych także odbywa się z wykorzystaniem bufora. Podczas wykonywania strumieniowej operacji czytania, gdy procedura zażąda danych np. z dysku, pobiera je nie bezpośrednio z dysku, ale z bufora. Podczas tej operacji stała ilość bajtów pobierana jest z dysku i umieszczana w buforze o tej samej długości. Następnie te dane są odczytywane z bufora. Po opróżnieniu bufora (nie ma w nim już ani jednego znaku) bufor jest automatycznie od nowa wypełniany następną porcją informacji. Wykorzystania mechanizmu buforowania istotnie przyspiesza działanie operacji wejścia/wyjścia, ponieważ maleje ilość odwołań do dysku. Programista musi zwrócić uwagę na fakt, że operacje obsługi bufora mogą być źródłem potencjalnych kłopotów, np. nie zawsze zawartość bufora zostanie faktycznie zapisana. Taki fakt nastąpi, gdy np. w programie wystąpi błąd.

Jest wiele możliwości obsługi operacji wejście/wyjście. Najczęściej używane w praktyce funkcje i makrodefinicje pokazane są w tabeli 4.1.

Tabela 4.1.Wybrane funkcje i makrodefinicje wejścia –wyjścia.

| Funkcja/makro | opis |
|---|---|
| getc() | czyta kolejny znak z pliku |
| getch() | czyta znak z klawiatury bez wyświetlania |
| getchar() | czyta kolejny znak z pliku **stdin** |
| getche() | czyta znak z klawiatury, wyświetla echo |
| gets() | czyta znaki z pliku **stdin** do zmiennej znakowej |
| getw() | czyta słowo (liczbę całkowitą) z pliku |
| printf() | zapisuje argumenty do pliku |
| putc() | zapisuje znak do pliku |
| putch() | zapisuje na ekranie pojedynczy znak bez buforowania |
| putchar() | zapisuje znak do pliku **stdout** |
| puts() | zapisuje ciąg znaków do pliku **stdout** |
| putw() | zapisuje słowo (liczbę całkowitą) do pliku |
| scanf() | czyta argument z pliku **stdin** |

## 4.4. Makrodefinicja getc()

Makrodefinicja **getc()** odczytuje pojedynczy znak z pliku otwartego dla buforowanego wejścia. Deklaracja i definicja struktury znajduje się w pliku **<stdio.h>.** Znak jest odczytywany z bieżącej pozycji w pliku. Makrodefinicja **getc()** zwraca odczytany znak, jako liczbę typu **int**. W wyniku działania zwrócony zostaje kod wczytywanego znaku albo **EOF,** jeżeli został osiągnięty koniec pliku lub wystąpił błąd. Polecenie **getc()** najczęściej stosujemy, aby odczytać dane z pliku pojedynczo znak po znaku. Typowe zastosowanie makra **getc()** pokażemy w następującym programie.

Listing 4.1. Odczytywanie znaku, makro getc().

```
/* makro getc() */
#include <stdio.h>
#include <conio.h>
int main()
{ char zn;
  printf("Wprowadz znak z klawiatury i naciśnij
          Enter: ");
  zn = getc(stdin);
  printf("Znak wprowadzony to: '%c'\n", zn);
  getche();   //przytrzymanie ekranu
  return 0;
}
```

Po uruchomieniu programu na ekranie pojawia się napis:

```
Wprowadz znak z klawiatury i nacisnij Enter :
```

Program czeka na wprowadzenie znaku. Jeżeli zostanie wprowadzona z klawiatury litera np. **a** to w dalszym ciągu program czeka na akcję ze strony użytkownika. Po naciśnięciu klawisza Enter, na ekranie pojawi się napis:

```
Znak wprowadzony to : 'a'
```

Użytkownik może także wprowadzić drugi znak np. **b** i nacisnąć klawisz Enter. Na ekranie pojawi się także napis:

```
Znak wprowadzony to : 'a'
```

Bez względu, co wprowadzimy po pierwszym znaku, program pobierze tylko pierwszy wprowadzony znak. W linii

```
zn = getc(stdin);
```

jako parametru użyliśmy predefiniowanego strumienia **stdin**, ponieważ deklaracja makra ma postać :

```
int getc(FILE *stream)
```

W nowoczesnym kompilatorze języka C zazwyczaj istniej 5 predefiniowanych strumieni, które są automatycznie otwierane, gdy jest uruchamiany program.

Predefiniowane strumienie:

| | |
|---|---|
| **stdin** | standardowe urządzenie wejściowe |
| **stdout** | standardowe urządzenie wyjściowe |
| **stderr** | standardowe urządzenie wyjściowe do obsługi błędów |
| **stdaux** | standardowe urządzenie dodatkowe (auxiliary) |
| **stdprn** | standardowa drukarka |

Makro **getc()** zalecane jest do odczytywania danych z pliku pojedynczo znak po znaku. Tego typu zastosowanie pokazano w następnym programie. Program służy do odczytania pojedynczej linii z pliku określonego przez użytkownika. W naszym przypadku chcemy wyświetlić pierwszą linię zapisaną (przykładowo) w pliku **autoexec. old.** Założono, że w linii będzie maksymalnie 80 znaków. Zastosowano też obsługę błędu otwarcia. Tego typu kontrola jest mocno zalecana w poprawnie pisanych programach. Jeżeli plik nie istnieje, lub ścieżka dostępu jest nieprawidłowo podana, program korzystając z funkcji **exit()** kończy działanie. Funkcja **exit()** wykorzystywana jest do normalnego zakończenia procesu, jednocześnie czyści bufor i zamyka pliki. Użycie funkcji **exit()** wymaga włączenia pliku nagłówkowego **<stdlib.h>** lub **<process.h>.**

Listing 4.2. Makro getc(), obsługa błędu otwarcia pliku

```
/* makro getc() */
#include <stdio.h>
#include <conio.h>
#include <process.h>     //dla exit()
int main()
{ FILE *plik;
  char buf[81];
  int i, c;
  if ((plik=fopen ("c:\\autoexec.old","r"))==NULL)
   { printf("problem z otwarciem\n");
     exit(0);
   }
  c = getc(plik);
  for(i=0;(i<80)&&(feof(plik)==0) && (c != '\n'); i++)
   { buf[i] = c;
     c = getc(plik);
   }
  buf[i] = '\0' ;
  printf("pierwsza linia pliku autoexec: \n");
  printf(" %s\n", buf);
  getche();   //przytrzymanie ekranu
  return 0;
}
```

Po uruchomieniu programu (zawartość wyświetlanego pliku jest przykładowa) na ekranie ukazuje się komunikat:

```
pierwsza linia pliku autoexec:
rem - By Windows Setup - C;WINDOWS\COMMAND\MSCDEX.EXE
/D:IDECD000
```

Oczywiście tego typu napis zależy od konkretnego komputera i konkretnego pliku **autoexec**. W linii

```
FILE *plik;
```

zadeklarowano wskaźnik do pliku, z którego ma być odczytywany znak.
Obsługa błędu otwarcia pliku umieszczona jest w następującym fragmencie:

```
if ((plik = fopen ("c:\\autoexec.old", "r")) == NULL)
{ printf("problem z otwarciem\n");
  exit(0);
}
```

Odczytywanie znaków umieszczonych w pierwszej linii pliku realizowane jest za pomocą pętli **for**:

```
c = getc(plik);
for (i=0; (i<80) && (feof(plik) == 0) &&(c != '\n'); i++)
   { buf[i] = c;
     c = getc(plik);
   }
```

W pętli **for** sprawdzane jest jednocześnie czy nie natrafiono na koniec pliku oraz czy nie natrafiono na koniec linii.

## 4.5. Makrodefinicja getchar()

Makrodefinicja **getchar()** odczytuje pojedynczy znak z pliku **stdin**. Plik **stdin** jest standardowo połączony z wejściem klawiatury. Deklaracja i definicja struktury znajduje się w pliku **<stdio.h>.** Znak jest odczytywany z bieżącej pozycji w pliku. Makro **getchar()** zwraca odczytany znak, jako liczbę typu **int**. W wyniku działania zwrócony zostaje kod wczytywanego znaku albo **EO**F, jeżeli został osiągnięty koniec pliku lub wystąpił błąd. Polecenie **getchar()** najczęściej stosujemy, aby odczytać pojedynczy znak z klawiatury. W przypadku większości kompilatorów wprowadzając znak możemy korzystać albo z typu **char,** albo z typu **int**. Dostępne są dwa formaty:

```
int c;
c = getchar();
```

albo też :

```
char c;
c = getchar(),
```

Typowe zastosowanie makra **getchar()** pokażemy w programie pokazanym na listingu 4.3. W pętli **while** pokazanego programu

```
while ((zn = getchar()) != '\n')
     printf("%c", zn);
```

realizowane jest wczytywanie znaków z klawiatury. Użytkownik wprowadza określoną ilość znaków a następnie naciska klawisz **Enter** kończąc wpisywanie znaków. Program wyświetli wprowadzone znaki. Zmiennej **zn,** która jest tutaj typu **int** jest przypisany wynik wykonania makrodefinicji **getchar().** Makrodefinicja **getchar()** nie ma argumentu.

Listing 4.3 Wczytywanie znaków z klawiatury, makro getchar().

```
/* makro getchar() */
#include <stdio.h>
#include <conio.h>     // dla getche()
int main()
{ int zn;
  printf(" dopoki nie nacisniesz Enter,
           beda wprowadzane znaki");
  printf("\nwprowadz znak z klawiatury : ");
  while ((zn = getchar()) != '\n')
  printf("%c", zn);
  getche();   //przytrzymanie ekranu
  return 0;
}
```

W następnym programie (listing 4.4), prezentujemy zastosowanie makra **getchar()** do wykonania konwersji małych liter do dużych. Znakom są przyporządkowane wartości całkowite. Np. znakowi '**a**' przyporządkowana jest wartość 97. Te wartości są uporządkowane w porządku alfabetycznym. Jeżeli to weźmiemy pod uwagę to wykonanie operacji typu:

```
'a' + 1
```

ma sens, ponieważ wartość tego wyrażenia wynosi 98, a tej wartości przyporządkowany jest znak '**b**'. Wyrażenie

```
'Z' - 'A'
```

ma wartość 25. Może też napisać:

```
'A' - 'a' = 'B' - 'b' = 'C' -'c' = .......itd.
```

W pętli **while** prezentowanego programu

```
while ((zn = getchar()) != EOF)
  if ('a' <= zn && zn <= 'z')
    putchar(zn + 'A' - 'a');
  else
    putchar(zn);
```

makro **getchar()** zmiennej **zn** przypisuje wartość pobranego znaku z klawiatury. W warunku **if** sprawdzane jest czy wprowadzany znak mieści się w zakresie liter alfabetu. Jeżeli jest to prawda wyrażenie w makrodefinicji **putchar()**:

```
zn + 'A' - 'a'
```

dokonuje konwersji litery małej w dużą. Makro **putchar()** wyświetla wynik.

Listing 4.4 Zamiana małych liter na duże, makro getchar()

```
/* makro getchar() */
#include <stdio.h>
#include <conio.h>     // dla getche()
int main()
{ int zn;
  printf("test getchar() - program konczy CTRL+Z \n");
  while ((zn = getchar()) != EOF)
    if ('a' <= zn && zn <= 'z')
       putchar(zn + 'A' - 'a');
    else
      putchar(zn);
  getche();   //przytrzymanie ekranu
  return 0;
}
```

Ten program nie ma prostego mechanizmu zatrzymania. Jeżeli użytkownik chce przerwać działanie programu musi wprowadzić polecenia **CTRL+Z** albo skorzystać z klawiszy **CTRL+Break.** Mało elegancka modyfikacja może polegać na zmianie warunku w pętli **while**. Jeżeli ustalimy, że wprowadzenie znaku 0 (zero) z klawiatury ma kończyć działanie to modyfikacja może mieć postać:

```
while ((zn = getchar()) != '0')
```

Biblioteki języka C zawierają makrodefinicję **toupper()** wykorzystywaną do zamiany małych liter w kodzie ASCII na duże.

## 4.6. Funkcje getch() i getche()

Funkcje **getch()** i **getche()** odczytują znaki z klawiatury bez buforowania. Funkcja **getch()** nie wyświetla znaku na ekranie, funkcja **getche()** - wyświetla znak na ekranie ( jest to tzw. ***echo***). Wywołanie tych funkcji jest następujące:

```
zn = getch();
```

lub

```
zn = getche();
```

Funkcje są typu **int,** nie mają parametru wywołania. Wymagają pliku nagłówkowego **<conio.h>.**

Następujące programy demonstrują zastosowanie omawianych funkcji.

Listing 4.5. Wprowadzanie znaków, getch() i getche()

```
#include <stdio.h>
#include <conio.h>
int main()
{ char zn;
  printf("Wprowadz znak z klawiatury : ");
  zn = getch();
  printf("\nwprowadzono znak '%c'\n", zn);
  getche();   //przytrzymanie ekranu
  return 0;
}
```

Po uruchomieniu programu, na ekranie pojawi się napis:

```
Wprowadz znak z klawiatury:
```

Program czeka na wprowadzenie znaku z klawiatury. Po wprowadzeniu znaku z klawiatury następuje zakończenie działania programu.
Następny program wykonuje podobne zadanie - zamienia małe litery na duże wykorzystując funkcję biblioteczną **toupper().** Program pozwala na czytanie znaków z klawiatury z dwoma opcjami zakończenia:, gdy napotka znak powrotu karetki **\r** lub gdy przeczyta 80 znaków.

Listing 4.6. Zamiana liter na duże, funkcja getche() i toupper()

```
/* makro getche() */
#include <stdio.h>
#include <conio.h>     // dla getche()
#include <ctype.h>     // dla toupper()
int main()
{ int i;
  char zn[81];
  printf("Wprowadz tekst z klawiatury : \n");
  for (i=0; i<80; i++)
   {
    if ((zn[i] = toupper(getche()))  == '\r')
      break;
   }
  zn[i] = '\0' ;
  printf("\nwprowadzono tekst : ");
  printf("\n%s", zn);
  getche();   //przytrzymanie ekranu
  return 0;
}
```

Wczytywanie znaków oraz warunek zakończenia po przeczytaniu 80 znaków realizuje pętla **for**:

```
for (i=0; i<80; i++)
{   if ((zn[i] = toupper(getche()))  == '\r')
    break;
}
```

Warunek zakończenia (dla powrotu karetki) sprawdza konstrukcja **if** :

```
    if ((zn[i] = toupper(getche()))  == '\r')
```

Funkcja **getche()** pobiera znak z klawiatury, jest ona parametrem funkcji **toupper().** Funkcja dokonuje konwersji i umieszcza znak w zmiennej tablicowe **zn[i].**

## 4.7. Funkcja gets()

Funkcja **gets()** służy do wprowadzania łańcucha do zmiennej. Ta funkcja czyta linię ze standardowego pliku wejściowego **stdin**. Plik **stdin** jest standardowo połączony z wejściem klawiatury. Deklaracja i definicja struktury znajduje się w pliku **<stdio.h>.** Parametrem funkcji jest nazwa zmiennej, użytkownik musi zarezerwować miejsce na tą zmienną. Funkcja **gets()** odczytuje i zapamiętuje w buforze znaki wprowadzane z klawiatury, aż do napotkania znaku nowej linii. Wtedy zastępuje ten znak znakiem **NULL**. Dla parametru funkcji **gets()** kompilator zarezerwuje tyle komórek pamięci ile zażąda programista. Ponieważ jedna komórka musi być przeznaczona na ogranicznik łańcucha, użytkownik może wprowadzić o jeden znak mniej, niż zadeklarowano. Wprowadzony łańcuch nie będzie wartością zmiennej dopóki nie zostanie naciśniety klawisz Enter. Podczas wpisywania łańcucha, na ekranie monitora pojawia się ***echo***. Polecenie **gets()** stosujemy wyłącznie do wprowadzania łańcuchów. Poniższy program wykorzystując funkcje **gets()** odczytując łańcuch wprowadzony z klawiatury.

Listing 4.7. Odczytywanie łańcucha, funkcja gets()

```
/* funkcja gets() */
#include <stdio.h>
#include <conio.h>     // dla getche()
int main()
{ char imie[81];
  puts("\n Napisz nazwisko i imie ");
  gets(imie);
  puts("Witamy! ");
  puts(imie);
  getche();   //przytrzymanie ekranu
  return 0;
}
```

W tym przykładzie użytkownik wprowadza z klawiatury swoje nazwisko i imię, program wyświetla ten łańcuch za pomocą funkcji **puts().** W linii:

```
char imie[81];
```

zarezerwowano dla zmiennej **imie** 81 znaków. Ponieważ jeden znak zarezerwowany jest na ogranicznik łańcucha, użytkownik może wprowadzić maksymalnie 80 znaków. Jeżeli zachodzi potrzeba wprowadzania liczb i łańcuchów z klawiatury, używa się jednocześnie funkcji **scanf()** i **gets().** Może to prowadzić do sporych kłopotów. Może się zdarzyć, że znak nowej linii może pozostać w buforze klawiatury po wykonaniu funkcji **scanf().** Funkcja **gets()** wczyta ten znak, czyli wczyta pusty łańcuch. Aby unikać tego typów kłopotów, rekomenduje się nie mieszać tych funkcji, a do wczytywania liczb i łańcuchów stosować wyłącznie **gets().** Zwykle wczytuje się liczbę, jako łańcuch a następnie stosuje się funkcje biblioteczne takie np. jak **atoi()** do konwersji łańcucha na liczbę. Przykład stosowania funkcji **gets()** do wczytywania zmiennych różnych typów pokazuje poniższy przykład.

Listing 4.8. gets() wczytuje zmienne różnych typów

```
/* funkcja gets() */
#include <stdio.h>
#include <conio.h>     // dla getche()
#include <stdlib.h>    // dla atoi(), atof()
int main()
{  char string[81];
  char imie[41];
  int nr_pacjenta;
  float waga,wzrost;
  printf("Napisz numer pacjenta ..... ");
  gets(string);
  nr_pacjenta = atoi(string);
  printf("Napisz nazwisko i imie pacjenta.... ");
  gets(imie);
  printf("Wprowadz wage pacjenta..... ");
  gets(string);
  waga = atof(string);
  printf("Wprowadz wzrost pacjenta..... ");
  gets(string);
  wzrost = atof(string);
  printf("\n Pacjent Nr %d o nazwisku : %s",
             nr_pacjenta,imie);
  printf("\n ma wage = %.2f kg oraz wzrost =
             %.2f cm", waga, wzrost);
  getche();   //przytrzymanie ekranu
  return 0;
}
```

W tym programie, wczytujemy z klawiatury trzy typy zmiennych: **char**, **int** i **float** a następnie stosujemy funkcje biblioteczne do konwersji. W linii:

```
nr_pacjenta = atoi(string);
```

zmiennej **nr_pacjenta** przypisana zostaje liczba całkowita, konwersji dokonuje funkcja **atoi().** W linii

```
waga = atof(string);
```

zmiennej **waga** przypisana zostaje liczba zmiennoprzecinkowa, konwersji dokonuje funkcja **atof().**
Biblioteka dostarcza wiele podobnych funkcji konwertujących. W tabelach 4.2 i 4.3 pokazano najczęściej używane do zamiany znaków funkcje biblioteczne.

Tabela 4.2. Zestaw funkcji zamieniających znak na liczby.

| Nr | Funkcja | Zamienia ciąg znaków : |
|---|---|---|
| 1 | atof() | na wartość zmiennoprzecinkową podwójnej precyzji |
| 2 | atoi() | na liczbę całkowitą |
| 3 | atol() | na liczbę całkowita podwójnej precyzji |
| 4 | strod() | na liczbę zmiennoprzecinkową podwójnej precyzji |
| 5 | strol() | na liczbę całkowitą podwójnej długości |
| 6 | stroul() | na liczbę całkowitą podwójnej długości bez znaku |

Tabela 4.3. Zestaw funkcji zamieniających liczby na znaki.

| Nr | Funkcja | Zamienia liczbę: |
|---|---|---|
| 1 | ecvt() | zmiennoprzecinkową podwójnej precyzji na ciąg znaków bez kropki dziesiętnej |
| 2 | ecvt() | zmiennoprzecinkową podwójnej precyzji na ciąg znaków, ustalamy liczbę cyfr |
| 3 | gcvt() | zmiennoprzecinkową podwójnej precyzji na ciąg znaków o określonej ilości cyfr |
| 4 | itoa() | całkowitą na ciąg znaków |
| 5 | ltoa() | całkowitą podwójnej długości na ciąg znaków |
| 6 | ultoa() | całkowitą podwójnej długości bez znaku na ciąg znaków |

## 4.8. Funkcja printf()

Do obsługi wyjścia w języku C, jedną z najbardziej rozbudowanych i najczęściej stosowanych jest funkcja biblioteczna **printf().** Jest ona używana do zapisu ciągu znaków a także wartości zmiennych, odpowiednio sformatowanych, do standardowego pliku wyjściowego **stdout**. Praktycznie oznacza to wyświetlanie danych na ekranie monitora. Deklaracja funkcji znajduje się w pliku nagłówkowym **<stdio.h>.** Formalnie składnia funkcji **printf()** ma postać:

```
printf (const char *łańcuch_formatujący, lista_danych)
```

Przykładowe wywołanie funkcji (przykład nie oddaje niezliczonych możliwości tej funkcji) może mieć postać:

```
printf("\nPole kwadratu o boku a %d wynosi %d", a,a*a);
```

W tym przykładzie napis:

```
"\nPole kwadratu o boku a %d wynosi %d"
```

jest ***łańcuchem formatującym. Listę danych*** tworzy napis zawierający dwie dane (jedną zmienną i jedno wyrażenie):

```
a, a*a
```

Lista danych zawiera wartości, stałe lub zmienne. Tą listę oddzielamy przecinkiem od łańcucha formatującego, listę danych też oddzielamy przecinkami. Możemy powiedzieć, że funkcja **printf()** oczekuje dwóch ogólnych argumentów - łańcucha formatującego, ujętego w parę cudzysłowów i listy argumentów poprzedzonych przecinkiem. Argument **łańcuch_formatujący** jest wymagany przy wywołaniu funkcji, formalnie argument **lista_danych** nie jest konieczna, tak jak w następującym przykładzie:

```
printf("To jest funkcja printf()");
```

Istotnym elementem łańcucha formatującego jest znak **%.** Jego brak informuje kompilator, że żaden argument nie jest oczekiwany, lista danych nie wystąpi, pojawienie się znaku **%** jest sygnałem dla kompilatora, że wystąpi ***specyfikator formatowania*** i należy oczekiwać listy danych. Specyfikator formatowania jest pojedynczym znakiem wyznaczającym typ drukowanej zmiennej. Każdy element z listy danych musi mieć specyfikator formatowania. Specyfikator formatowania **%d** informuje kompilator, że będzie drukowana wartość typu **int.** Ogólna postać specyfikatora formatu jest następująca:

```
%[znacznik][szerokość][precyzja][tryb_adresowania]
 [rozmiar][typ]
```

wymagany jest symbol **%** oraz [**typ**], inne elementy (pola) są opcjonalne.

**Typ.**

Pole **typ** informuje kompilator o typie zmiennej. Gdy kompilator tworzy kod wynikowy, wstawia daną z listy danych w miejsce zajmowane przez znak symbolizujący typ. W przykładzie:

```
printf("%d",13);
```

gdy wykonywana jest ta instrukcja, wartość 13 jest wstawiana w miejsce **d**, i wyświetlany jest wynik 13. Znak **%** a także cudzysłowy nie są wyświetlane. Ten sam skutek otrzymamy, gdy użyjemy zmiennej tak, jak w przykładzie:

```
int nr;
nr = 13;
printf("%d",nr);
```

W tabeli 4.4 pokazano listę formatów realizowanych za pomocą funkcji **printf().**

Tabela 4.4. Spis możliwych formatów realizowanych przez funkcje **printf()**.

| Symbol | Typ | Format wyjściowy |
|---|---|---|
| d | int | Liczba całkowita dziesiętna ze znakiem lub bez |
| i | int | Liczba całkowita dziesiętna ze znakiem lub bez |
| u | unsigned | Liczba całkowita bez znaku |
| o | unsigned | Liczba całkowita ze znakiem w postaci ósemkowej |
| x | unsigned int | Liczba całkowita bez znaku w postaci szesnastkowej, małe litery |
| X | unsigned int | Liczba całkowita bez znaku w postaci szesnastkowej, duże litery |
| f | double lub float | Liczba zmiennoprzecinkowa ze znakiem w formacie : [znak][cyfry].[cyfry] np. -123.456700 |
| e | double lub float | Liczba zmiennoprzecinkowa ze znakiem w zapisie naukowym w postaci : [znak][cyfry].[cyfry]e[znak][cyfry] Np. -1.234567e+002 |
| E | double lub float | Liczba zmiennoprzecinkowa ze znakiem w zapisie naukowym w postaci : [znak][cyfry].[cyfry]E[znak][cyfry] Np. -1.234567E+002 |
| g | double lub float | Liczba zmiennoprzecinkowa ze znakiem drukowana zgodnie z formatem e lub f, wybierany jest optymalny sposób wyświetlania . |
| G | double lub float | Liczba zmiennoprzecinkowa ze znakiem drukowana zgodnie z formatem g, zamiast e drukowane jest E |
| c | char | Pojedynczy znak, printf("%c",'A') drukuje literę A |
| s | wskazanie na char | Ciąg znaków (łańcuch) |
| n | wskazanie na int | Limitowana liczba znaków, nie jest to format. Argumentem jest wskaźnik do liczby całkowite. |
| p | odległe wskazanie na void | Drukowany jest adres w postaci SSSS:OOOO, SSSS jest adresem segmentu a OOOO jest przemieszczeniem. Liczba w postaci szesnastkowej |

W łańcuchu sterującym specyfikator formatu może być umieszczony w dowolnym miejscu.

W przykładzie:

```
printf("To jest %d wywołanie funkcji",13);
```

łańcuchem sterującym jest napis:

```
"To jest %d wywołanie funkcji"
```

a wynikiem wykonania tej instrukcji jest napis na ekranie monitora:

```
To jest 13 wywołanie funkcji
```

Jeżeli chcemy, aby był wydrukowany znak % należy go napisać dwukrotnie.

Listing 4.9. Formaty drukowania, funkcja printf()

```
/* funkcja printf() */
#include <stdio.h>
#include <conio.h>     // dla getche()
int main()
{ int f1 = 19999;
  int f2 = -123;
  double f3 = 123.45678;
  char f4 = 'C';
  char f5[] = "Ilustrujemy formaty";
  printf("\n format %%d  %d",f1);
  printf("\n format %%i  %i",f2);
  printf("\n format %%u  %u",f1);
  printf("\n format %%o  %o",f1);
  printf("\n format %%x  %x",f1);
  printf("\n format %%X  %X",f1);
  printf("\n format %%f  %f",f3);
  printf("\n format %%e  %e",f3);
  printf("\n format %%E  %E",f3);
  printf("\n format %%g  %g",f3);
  printf("\n format %%G  %G",f3);
  printf("\n format %%c  %c",f4);
  printf("\n format %%s  %s",f5);
  printf("\n To byl test printf() - program konczy
          ENTER");
  getche();   //przytrzymanie ekranu
  return 0;
}
```

Początkujący programista powinien poeksperymentować z funkcją **printf().** Ma ona wiele możliwości. Należy jednak pamiętać, że jest to funkcja biblioteczna, może ona zawierać wiele elementów niestandardowych.
Należy tez zwrócić uwagę na sprawę dokładności obliczeń i precyzji wydruku. Gdy dane wprowadzone są z dokładnością do jednego miejsca po przecinku, nie ma sensu żądanie wydruku z np. dziesięcioma cyframi znaczącymi po kropce dziesiętnej. Należy pamiętać, że dokładność kosztuje - zwiększa się czas

obliczeń i ilość zajętej pamięci. Znaczniki typu są bardzo rozbudowane.
Program 4.9 demonstruje skutki wywołania funkcji **printf()** - zastosowano wszystkie przytoczone w tabeli formaty z wyjątkiem formatu p. Wybrano określone typy zmiennych:

```
int f1 = 19999;
int f2 = -123;
double f3 = 123.45678;
char f4 = 'C';
char f5[] = "Ilustrujemy formaty";
```

i zażądano wydruku tych wartości tych zmiennych w wyspecyfikowanych formatach.

Listing 4.10. Specyfikator formatu p, funkcja printf().

```
#include <stdio.h>
#include <conio.h>     // dla getche()
char far *strf = "Odlegly ciag znakow...";
char near *strn = "Bliski ciag znakow...";
double far *wsk1;
double near *wsk2;
int main()
{ double f1 = 1.2345;
  double f2 = 9.8765;
  wsk1 = &f1;
  wsk2 = &f2;
  printf(" Zmienna string segment:przemieszczenie\n");
  printf("'odlegly' ciag znakow   %Fp\n",
     (void far *) strf);
  printf("'bliski'  ciag znakow   %Fp\n",
     (void far *) strn);
  printf(" \nZmienna double
     segment:przemieszczenie\n");
  printf("f1= %f           %Fp\n", f1,
          (void far *) wsk1);
  printf("f2= %f           %Fp\n", f2,
          (void far *) wsk2);
  printf("\n To byl test formatu p -
     program konczy ENTER");
  getche();   //przytrzymanie ekranu
  return 0;
}
```

Format p jest rzadziej używanym. Dzięki temu formatowi drukowany jest pełny adres zmiennej, tzn. adres segmentu i jego przemieszczenie. W modelu pamięci **small** lub **medium** ( jest to typowe dla PC-tów typu IBM) argument powinien być przekazywany jako **(void far *)**. Jeżeli w formacie zastosujemy znacznik N:

```
%Np.
```

to drukowane będzie tylko przesuniecie adresu. Zwracamy uwagę, że ten format jest niestandardowy, dlatego nie należy go nadużywać. Do demonstracji tego formatu wybrano następujące typy (program z listingu 4.10):

```
char far *strf = "Odlegly ciag znakow...";
char near *strn = "Bliski ciag znakow...";
double far *wsk1;
double near *wsk2;
```

**Znacznik**

Aby zastosować opcję [znacznik] możemy użyć następujących znaków:

| | |
|---|---|
| + | dla wartości numerycznej, w zależności od sytuacji drukowany jest znak + lub - znak |
| - | podczas drukowania ciąg znaków będzie wyrównany lewostronnie dodatnie wartości numeryczne są poprzedzone pustymi spacjami |
| # | przy drukowaniu wartości zmiennych ósemkowych lub szesnastkowych, drukowana wartość jest poprzedzana oznaczeniem o, 0x lub 0X. Dla formatu e, E oraz f wymuszone jest drukowanie kropki dziesiętnej. Dla formatu g oraz G znacznik # drukuje kropkę dziesiętną i wszystkie nieznaczące zera. |

**Szerokość**

Opcja [szerokość] pozwala na ustalenie całkowitej szerokości pola przeznaczonego na wydrukowanie zmiennej. Pozwala to na tabulowane przedstawianie wyników. W przypadku konfliktu następuje obcinanie lub uzupełnianie znaków.

**Precyzja**

Opcja [precyzja] pozwala ustalić ilość miejsc występujących po kropce dziesiętnej podczas drukowania wyników. W tym polu musi być nieujemna liczba całkowita. Gdy w tym polu zostanie wstawiony znak * precyzja określana jest w czasie działania programu. Funkcja **printf()** różnie reaguje na tą opcję. Należy sprawdzić dokumentację. Dla formatu np. c, pole precyzji jest ignorowane, a dla formatu s, precyzja określa maksymalną liczbę znaków, jakie będą drukowane. Może się zdarzyć, że funkcja **printf()** obetnie nadmiarowe znaki - należy ostrożnie projektować to pole.

**Tryb adresowania**

Ta opcja jest specyficzna dla konkretnego kompilatora języka C, przesłania domyślne tryby adresowania w modelu pamięci. Można użyć albo znaku F (dla

far - daleki) albo N (dla near - bliski). Ma to zastosowanie jedynie w przypadku przekazywania do funkcji printf() zmiennej typu wskaźnikowego.

**Rozmiar**

Opcja [rozmiar] pozwala modyfikować pole [typ]. Można stosować znaki h, l, oraz L.

- **h** oznacza przekazanie danej typu **short int**. Stosujemy ten znacznik podczas drukowania liczb całkowitych (formaty : d, i, o, x, X )
- **l** oznacza przekazanie danej typu **long int**. Stosujemy ten znacznik podczas drukowania zmiennych typu **int** oraz **unsigned int** (formaty: d, i, o, x, X ). Znacznik l stosowany jest także dla liczb zmiennoprzecinkowych (formaty : e, E, g, G ) dla określenia zmiennej typu **double** zamiast **float**.
- **L** Znacznik stosowany przy drukowaniu wartości zmiennoprzecinkowych typu **long double** (formaty: e, E, f, g, G )

**Znaki specjalne**

Występujący w łańcuchu formatującym znak końca linii **\n** jest przykładem znaku specjalnego. Język C używa kilku znaków specjalnych. Znak specjalny zawsze poprzedzony jest znakiem \ (***backslash***).

| Sekwencja znaków | wartość ASCII | znaczenie |
|---|---|---|
| \a | 7 | sygnał dźwiękowy (BEL) |
| \b | 8 | cofniecie o 1 znak (BS, backspace) |
| \t | 9 | tabulacja pozioma (HT) |
| \v | 11 | tabulacja pionowa (VT) |
| \n | 10 | nowa linia (LF) |
| \f | 12 | przejście do następnej strony (FF) |
| \r | 13 | przejście do początku wiersza (CR) |
| \" | 34 | cudzysłów |
| \' | 39 | apostrof |
| \? | 63 | znak zapytania |
| \\ | 92 | pojedynczy znak \ (backslash) |
| \0 | 0 | znak pusty (null) |
| \xdd | kod ASCII w systemie szesnastkowym (znak **d** oznacza cyfrę) | |
| \ddd | kod ASCII w systemie ósemkowym (znak **d** oznacza cyfrę) | |

## 4.9. Makrodefinicja putc()

Makrodefinicja **putc()** zapisuje pojedynczy znak do pliku otworzonego dla buforowanego wyjścia. Stosowanie tego makra wymaga pliku nagłówkowego **<stdio.h>.** Plik, do którego ma być zapisany znak musi być otwarty.

Plik otwieramy standardowo za pomocą funkcji **fopen()** lub **freopen()** albo zapisujemy znak do pliku, który otwierany jest automatycznie dla każdego programu tzn. **stdout** lub **stderr**. Składnia tego makra ma następującą postać:

```
int putc(int c,  FILE *wskaźnik_pliku)
```

## 4.10. Funkcja scanf()

Do obsługi wejścia w języku C jedną z najbardziej rozbudowanych jest funkcja biblioteczna **scanf().** Jest ona używana do odczytywania ciągów znaków ze standardowego pliku wejściowego **stdin** i przekształcaniu ich na wartości zmiennych, zgodnie z podanym formatem. Praktycznie oznacza to wczytywanie danych z klawiatury. Funkcja śledzi (skanuje) klawiaturę, badając naciśnięte klawisze. Możemy wprowadzać dane numeryczne, łańcuchowe i typu **char**. Deklaracja funkcji znajduje się w pliku nagłówkowym **<stdio.h>.** Formalnie składnia funkcji **scanf()** ma postać:

```
scanf(const char *łańcuch_formatujący, lista_danych)
```

a przykładowe wywołanie funkcji (przykład nie oddaje niezliczonych możliwości tej funkcji) może mieć postać:

```
scanf(" %d", &bok_a);
```

W tym przykładzie napis:

```
" %d "
```

jest łańcuchem formatującym. Listę danych tworzą adresy zmiennych (tutaj jeden):

```
&bok_a
```

Symbol **&** reprezentuje operator adresu. W przykładowej instrukcji występuje format **%d,** który powoduje, że funkcja **scanf()** interpretuje znaki wejściowe, jako liczby dziesiętne całkowite i umieszcza wartości pod adresem zmiennej **bok_a.** Gdy używamy klawiatury do wprowadzania danych, sekwencja znaków jest drukowana na ekranie monitora i sekwencja znaków jest przyjmowana przez program. Ta sekwencja nosi nazwę ***strumienia wejściowego***. Lista danych zawiera zmienne. Tą listę oddzielamy przecinkiem od łańcucha formatującego, listę danych też oddzielamy przecinkami. Każda zmienna z listy danych, której adres jest podany musi mieć ***specyfikator formatowania.***
Ogólna postać specyfikatora formatu jest następująca:

```
%[*][szerokość][tryb_adresowania][rozmiar][typ]
```

wymagany jest symbol **%** oraz [**typ**], inne elementy (pola) są opcjonalne.

Tabela 4.5. Spis możliwych formatów realizowanych przez funkcje **scanf().**

| Symbol | Wskaźnik na typ | Format wejściowy |
|---|---|---|
| d | int | Liczba całkowita dziesiętna |
| l | long int | Liczba całkowita dziesiętna |
| i | int | Liczba całkowita dziesiętna, ósemkowa lub szesnastkowa |
| i | long | Liczba całkowita dziesiętna, ósemkowa lub szesnastkowa |
| u | unsigned | Liczba całkowita dodatnia |
| ul | unsigned long | Liczba całkowita dodatnia |
| o | unsigned | Liczba całkowita bez znaku w postaci ósemkowej |
| ol | long | Liczba całkowita bez znaku w postaci ósemkowej |
| x | unsigned int | Liczba całkowita bez znaku w postaci szesnastkowej, |
| xl | unsigned long | Liczba całkowita bez znaku w postaci szesnastkowej, |
| f | float | Liczba zmiennoprzecinkowa ze znakiem w formacie [znak][cyfry].[cyfry], np. -123.456700 |
| e | float | Liczba zmiennoprzecinkowa ze znakiem w zapisie naukowym w postaci [znak][cyfry].[cyfry]e[znak][cyfry] np. -1.234567e+002 |
| E | float | Liczba zmiennoprzecinkowa ze znakiem w Zapisie naukowym w postaci [znak][cyfry].[cyfry]E[znak][cyfry] np. -1.234567E+002 |
| g | float | Liczba zmiennoprzecinkowa ze znakiem drukowana zgodnie z formatem e lub f, |
| G | float | Liczba zmiennoprzecinkowa ze znakiem drukowana zgodnie z formatem g, |
| c | char | Pojedynczy znak, odczytuje białe znaki, |
| s | char tablica[] | Ciąg znaków (łańcuch) |
| n | | Nie jest to format odczytu. Argumentem jest wskaźnik do liczby całkowitej. |
| p | far *<br>near * | Drukowany jest adres w postaci SSSS:OOOO, SSSS jest adresem segmentu a OOOO jest przemieszczeniem. Liczba w postaci szesnastkowej |

Początkujący programista powinien poeksperymentować z funkcją **scanf().** Ma ona wiele możliwości. Należy jednak pamiętać, że jest to funkcja biblioteczna, może ona zawierać wiele elementów niestandardowych. Szczególną uwagę należy zwrócić na modyfikatory trybu adresowania **N** i **F**.

**Typ**

Pole [**typ**] informuje kompilator o typie zmiennej, to pole jest wymagane. Znaczniki typu są bardzo rozbudowane. W tabeli pokazano listę formatów realizowanych za pomocą funkcji **scanf().**
Wbrew pozorom, stosowanie funkcji **scanf()** nastręcza pewne kłopoty. Firma Borland, producent kompilatora Turbo C++ zamieściła stosowny komentarz w systemie pomocy.

> *Informacja producenta kompilatora Turbo C++ :*
> *A ...scanf function might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely.*
> *WARNING: scanf often leads to unexpected results if you diverge from an expected pattern. You must teach scanf how to synchronize at the end of a line.*
> *The combination of gets or fgets followed by sscanf is safe and easy, and therefore recommended over scanf.*

Poniższy program (pokazany na listingu 4.11) demonstruje skutki wywołania funkcji **scanf()** - zastosowano wybrane, proste formaty. Wybrano określone typy zmiennych:

```
int calkowita;
float rzeczywista;
char litera;
char napis[30];
float kwota;
```

Znaki wpisywane do strumienia wejściowego są przechowywane w buforze. Jeżeli funkcja **scanf()** nie wczyta wszystkich wprowadzonych znaków (a to może się zdarzyć), to niewprowadzone znaki pozostaną w buforze. Ponowne wywołanie funkcji **scanf()** spowoduje wprowadzenie do programu znaków pozostawionych w buforze, a niewprowadzanych aktualnie z klawiatury. Dobrym obyczajem jest takie pisanie programu, aby użytkownik wprowadzał jeden element danych a także poleca się umieszczenie informacji, jaki format danych jest oczekiwany.

Listing 4.11. Działanie funkcji scanf()

```
/* funkcja scanf() */
#include <stdio.h>
#include <conio.h>     // dla getche()
int main()
{ int calkowita;
  float rzeczywista;
  char litera;
  char napis[30];
  float kwota;
  printf("po wprowadzeniu danych nacisnij Enter\n");
  printf(" Wprowadz liczbe calkowita x1= ");
  scanf(" %d", &calkowita);
  printf("Wprowadz liczbe rzeczywista x2= ");
  scanf("%f", &rzeczywista);
  printf("Wprowadz litere zn= ");
  scanf(" %c", &litera);
  printf("Wprowadz napis nap= ");
  scanf("%20s", napis);
  printf("\nWyniki\nx1= %d  x2= %8.4f \nzn=  %c
      napis= %s", calkowita,rzeczywista,litera,napis);
  printf("\n\npodaj kwote w formacie:$44.44
     kwota= ");
  scanf(" $%f", &kwota);
  printf("kwota w dolarach to: $%.2f",kwota);
  getche();   //przytrzymanie ekranu
  return 0;
  }
```

**Opcje w specyfikacji formatu dla funkcji scanf().**

**Gwiazdka**

Opcja [*] oznacza zignorowanie pola wejściowego, znaki są odczytywane zgodnie z formatem, ale wartości nie są zapisywane, nie ma potrzeby podawania argumentu.

**Szerokość**

Opcja [szerokość] pozwala na ustalenie maksymalnej ilości przeznaczonych do wczytywania znaków. Pozwala to nakładanie ostrego warunku na ilość znaków, które będą wczytane.

**Tryb adresowania**

Ta opcja jest specyficzna dla kompilatora Turbo C++, przesłania domyślne tryby adresowania w modelu pamięci. Można użyć albo znaku F (dla far - daleki) albo N (dla near - bliski).

**Rozmiar**

Opcja [rozmiar] pozwala modyfikować pole [typ] . Można także stosować znaki h, l, oraz L.

**h** oznacza odczytywanie danej typu **short int**. Stosujemy ten znacznik podczas odczytywania liczb całkowitych (formaty: **o, x, X** )

**l** oznacza odczytywanie danej typu **long int.** Stosujemy ten znacznik podczas odczytywania zmiennych typu **int** oraz **unsigned int** (formaty: **d, i, o, x, X** ). Znacznik **l** stosowany jest także dla liczb zmienno – przecinkowych (formaty : **e, E, g, G** ) dla określenia zmiennej typu

# ROZDZIAŁ 5

## INSTRUKCJE WYBORU

## 5.1. Wstęp

Bardzo ważną cechą programów komputerowych jest możliwość podejmowania decyzji. Na podstawie decyzji, program komputerowy wybiera odpowiednie działanie. Dzięki temu mamy możliwość realizacji skomplikowanych zadań, w zależności od występujących okoliczności. W omawianych już pętlach, sprawdzane było wyrażenie warunkowe, występujące w pętli. Możliwe było kontrolowani ilości powtórzeń, a gdy wyrażenie warunkowe przestało być prawdziwe, nastąpiło „wyjście" z pętli. W tym rozdziale omówimy najważniejsze zagadnienia związane z instrukcjami wyboru.

## 5.2. Instrukcje sterujące: instrukcje wyboru

W prostych programach mamy do czynienia z sekwencyjnym wykonywaniem instrukcji: działania są podejmowane w kolejności pojawiania się instrukcji, program zaczyna się od pierwszej instrukcji i kończy działanie po wykonaniu ostatniej. W bardziej zaawansowanych programach wymagamy, aby wykonanie poszczególnych działań zależało od danych i typu obliczeń. Klasycznym przykładem jest obliczanie pierwiastków równania kwadratowego – w zależności od parametrów równania mamy różne wzory do wyliczania pierwiastków. Aby efektywnie wykonać rozwiązać zadanie, musimy dysponować konstrukcjami decyzyjnymi. Język C dostarcza trzy konstrukcje wyboru:

- instrukcję if
- instrukcję switch
- operator wyboru (warunkowy)

## 5.3. Operatory w wyrażeniach warunkowych.

Załóżmy, że chcemy napisać program, który podejmie akcję, gdy naciśnięty jest odpowiedni klawisz i nic nie zrobi (czeka), gdy naciśniemy inny. Tego typu program pokazany jest na listingu 5.1. Po uruchomieniu programu, na ekranie monitora mamy napis:

```
Program czeka na wlasciwy klawisz
Nacisnij dowolny klawisz
```

Jeżeli naciśniemy klawisz " **h** " to na ekranie pojawi się komunikat:

```
 Poprawnie,nacisnieto h klawisz
 Nacisnij Enter aby skonczyc
```

Jeżeli naciśniemy dowolny klawisz, który nie jest klawiszem " **h** " to na ekranie otrzymamy napis:

```
Nie ten klawisz
Nacisnij Enter aby skonczyc
```

Listing 5.1. Wyrażenia warunkowe, konstrukcja if

```
/* program demonstruje warunek */
#include <stdio.h>
#include <conio.h>
#define KL 'h'
intmain()
{ char znak;
  printf("\nProgram czeka na wlasciwy klawisz");
  printf("\nNacisnij dowolny klawisz");
  znak = getche();
    if (znak == KL)
    printf("\nPoprawnie,nacisnieto %c klawisz ",znak);
  else printf("\nNie ten klawisz");
    printf("\nNacisnij Enter aby skonczyc");
  getch();
 return 0;
}
```

Analizując program widzimy, że w linii:

```
#define KL 'h'
```

użyliśmy dyrektywy preprocesora #**define**. Zadaniem tej dyrektywy jest przypisanie nazwy **KL** stałej (u nas stałej znakowej ‘h’). Preprocesor przegląda cały program i w każdym miejscu, gdzie znajdzie napis **KL** wstawia znak ‘**h**’. Napis **KL** jest ***identyfikatorem***, napis ‘**h**’ jest tekstem. Spacja oddziela identyfikator od tekstu. Przyjęło się, że identyfikatory piszemy dużymi literami. Stosowanie dyrektywy #**define** jest bardzo wygodne. Gdy zechcemy, aby w naszym programie żądana reakcja nastąpiła po naciśnięciu klawisza np. ”**s**” to wystarczy zamieć znak '**h**' znakiem '**s**' w dyrektywie preprocesora. Unikamy przeglądania programu i szukania miejsca, gdzie jest znak ‘**h**’. W naszym przypadku ten znak występuje tylko w jednym miejscu – ewidentny zysk mamy gdyby wystąpił on w wielu miejscach. Zasadnicza część programy zawarta jest w następującym fragmencie:

```
znak = getche();
if (znak == KL)
    printf("\nPoprawnie,nacisnieto %c klawisz ",znak);
else printf("\nNie ten klawisz");
```

Wykonanie tego fragmentu jest następujące:

- za pomocą funkcji **getche()** pobierany jest znak z klawiatury
- jeżeli pobrany znak jest równy znakowi **h** to uruchamiana jest funkcję **printf ()** z komunikatem „Poprawnie naciśnięto .....”
- jeżeli pobrany znak nie jest równy znakowi **h** to uruchamiana jest funkcję **printf()** z komunikatem „Nie ten klawisz”

Funkcja **getche()** czyta pojedynczy znak w momencie naciśnięcia klawisza, bez czekania na klawisz Enter. Mamy podobną funkcję wejścia **getch()**, która nie wypisuje znaku na ekranie. W języku C mamy także funkcję **getchar(),** ale jest to tzw. funkcja buforowana, znak nie jest przekazany, dopóki nie będzie naciśnięty klawisz Enter. W pokazanym fragmencie została zastosowana konstrukcja **if else**, dzięki tej instrukcji, program ***podejmuje decyzję***. Podjęcie decyzji następuje po sprawdzeniu warunku:

```
if (znak == KL)
```

W języku C dysponujemy następującymi operatorami porównania:

| | |
|---|---|
| == | równe |
| != | różne |
| < | mniejsze |
| > | większe |
| <= | mniejsze lub równe |
| >= | większe lub równe |

Te operatory są operatorami binarnymi, łączone są lewostronnie. Wyrażenia z tymi operatorami mają swoją wartość:

```
int wartość 0
```

lub

```
int wartość 1
```

Spowodowane jest to faktem, że w języku C fałsz (ang. ***false***) jest reprezentowany przez wartość zero a prawda (ang. ***true***) reprezentowana jest przez każdą niezerową wartość. Wobec tego wartością **fałsz** może być 0 lub 0.0, a wartością dla **prawda** może być każda wartość różna od 0 lub 0.0. W wyrażeniach możemy używać kilku operatorów porównania, ale należy zdawać sobie sprawę, jaki będzie wynik. Przykłady legalnych wyrażeń to:

```
c == 'x'
x != -13.13
a + 13.0 * b != 1.1 / c
```

Rozważmy następujące wyrażenie (przypuszczamy, że programista chciał sprawdzić czy *x* należy do przedziału, czy nie):

```
3 < x < 5
```

Z matematycznego punktu widzenia, dla **x = 4** wyrażenie jest prawdziwe, a dla **x = 7** wyrażenie jest fałszywe. Rozważmy program pokazany na listingu 5.2, który sprawdza, czy podana liczba leży w ustalonym przedziale.

Listing 5.2. Operator porównania

```
/* operatory porownania */
#include <stdio.h>
int main()
{ int x,wynik;
  printf("\nPodaj liczbe :");
  scanf("%d",&x);
  wynik = 3 < x < 5;
  printf("wynik = %d",wynik);
  return 0;
}
```

Jeżeli uruchomimy program i wprowadzimy liczbę 4 to otrzymamy komunikat:

```
wynik = 1
```

Otrzymaliśmy poprawny wynik, prawdą jest, że 4 należy do przedziału (3,5).
Po wprowadzeniu wartości np. 150 otrzymamy komunikat:

```
wynik = 1
```

Wynikiem testu jest prawda, ale czy o to chodziło programiście? Wyjaśnienie otrzymanego wyniku dla wprowadzonej liczby 150 jest proste. Pamiętamy, że operator jest łączony lewostronnie, tzn., że wyrażenie

```
3 < x < 5
```

jest równoważne wyrażeniu

```
(3 < x ) < 5
```

Wyrażenie 3 <150 jest prawdziwe, ma wartość 1. Wobec tego:

```
(3 < x) < 5    jest równoważne  wyrażeniu     1 < 5
```

Wyrażenie 1 < 5 jest prawdziwe, ma wartość 1 i taki wynik jest drukowany. Jeżeli chcemy zaprogramować powyższy test, to raczej musimy napisać:

```
3 < x  &&  x < 5
```

Symbol **&&** jest logicznym operatorem **AND.**
W konstrukcji **if** bada się wyrażenia, aby podjąć odpowiednie działanie. Rozważmy następującą instrukcję:

```
if (x> 0) printf(”x jest dodatnie”);
```

Jeżeli **x** ma wartość 5 to wtedy (**x > 0**) jest prawdziwe i funkcja **printf()** będzie wywołana, jeżeli **x** ma wartość -3 to wtedy (**x > 0**) jest fałszem i funkcja **printf()** nie będzie wywołana. Zatem funkcja **printf()** jest wywoływana dla każdej niezerowej wartości wyrażenia w nawiasach. Możemy napisać:

```
if (x) printf(”x jest różne od zera”);
```

Wyrażenie warunkowe możemy zanegować za pomocą ***operatora negacji***. Mamy równoważne zapisy:

```
if (x==0) printf("x jest równe zeru");
if (!x) printf("x jest równe zeru");
```

Jeżeli założymy, że

```
x = 1
y = 2
```

to

```
x != y    jest prawdą
x == y  jest fałszem
```

Należy również mieć na uwadze to, że porównanie liczb rzeczywistych może mieć nieoczekiwany skutek. Często mamy do czynienia z wyrażeniem:

```
x < x + y
```

Formalnie jest to równoważne wyrażeniu

```
y > 0
```

Jeżeli **y** jest dodatnie to wyrażenie jest logicznie prawdziwe. W komputerze, jeżeli **x** jest dużą zmienną typu **float** a **y** jest bardzo małą liczbą rzeczywistą też typu **float**, to często może się zdarzyć, że wyrażenie

```
x < x + y
```

jest fałszywe. Powodowane jest to faktem, że powyższe wyrażenie jest równoważne wyrażeniu:

```
(x - (x + y)) < 0.0
```

a wartość tego wyrażenia zależy od systemu (ważne jest tzw. ***zero maszynowe***, dokładność maszynowa). Może zdarzyć się, że z dokładnością do zera maszynowego wartości **x** oraz **x + y** są równe, wartością logiczną wyrażenia będzie 0, czyli fałsz.
W wyrażeniach często potrzebujemy ***operatorów logicznych***. W języku C mamy trzy takie operatory:

| | |
|---|---|
| && | logiczne AND (iloczyn) |
| \|\| | logiczne OR ( alternatywa) |
| ! | logiczne NOT (negacja) |

***Logiczna negacja*** może być zastosowana do dowolnego wyrażenia. Jeżeli wyrażenie ma wartość logiczną 0 to negacja da wartość logiczną 1. Jeżeli wyrażenie ma niezerową wartość, to negacja da wartość logiczną 0.

Legalne wyrażenia to:

```
!5
!'x'
!((m +2)<=z)
a && b
x || !13.13
i < j  &&  x < y
'A' <=b  &&  b <= 'Z'
```

Operator **!** jest operatorem unarnym, łączonym prawostronnie. Wobec tego wyrażenie:

```
!!13     jest równoważne wyrażeniu    !(!13)
```

to znaczy wyrażenie !(!13) jest równoważne wyrażeniu !(0), którego wartością jest 1. Priorytet operatora **&&** jest wyższy niż operatora **||**. Oba te operatory mają niższy priorytet niż wszystkie operatory unarne, arytmetyczne i porównania. Łączność operatorów **&&** oraz **||** jest lewostronna.
Poniższe przykłady pokazują równoważne wyrażenia.

```
a && b && c          (a && b)  && c
a && b || c –13      (a && b) || (c – 13)
a || b && c – 13     a || (b && (c – 13))
```

## 5.4. Instrukcja if

Wyrażenia w programie są wykonywane kolejno, w porządku, w jakim programista je napisał (wykonywanie sekwencyjne). Często się jednak zdarza, że algorytm wymaga, aby kolejność wykonywanych wyrażeń była zmieniona - chcemy np. żeby pewien ciąg wyrażeń nie był wykonany. Tego typu działania nazywamy ***przekazywaniem sterowania***. Przekazywanie sterowania może być zrealizowane za pomocą konstrukcji **if.** Struktura wyboru **if** jest ***pojedynczą strukturą wyboru***, ponieważ wybiera ona lub ignoruje pojedyncze działanie. Przekazywanie sterowania może być realizowane także za pomocą konstrukcji **if....else**, jest to ***podwójna struktura wyboru***. Struktura wyboru **if** jest używana do wyboru pośród alternatywnych ciągów działań.
Instrukcja warunkowa **if** ma następującą postać:

```
if  (wyrażenie)
       instrukcja1
       instrukcja_następna
```

Jeżeli wyrażenie jest prawdziwe, wykonywana jest **instrukcja1**, jeżeli wyrażenie nie jest prawdziwe (fałsz), **instrukcja1** jest omijana i wykonuje się **instrukcja_następna**. W zasięgu konstrukcji **if** może występować wiele instrukcji – musimy wtedy określić blok, wykorzystując nawiasy klamrowe.

Pełna postać instrukcji warunkowej ma postać:

```
if  (wyrażenie)
{
   instrukcja1
   instrukcja2
         ...
         ...
   instrukcjaN
}
instrukcja_następna
```

Jeżeli **wyrażenie** jest prawdziwe, wykonywany jest ***blok instrukcji*** (**instrukcja1**, **instrukcja2,** ...., **instrukcjaN**), jeżeli wyrażenie jest fałszywe, blok instrukcji jest omijany i wykonuje się **instrukcja_następna**. Klasyczne zastosowanie instrukcji warunkowej **if** pokazuje program z listingu 5.3. Program oblicza pierwiastki równania kwadratowego. Zanim zanalizujemy program omówimy pewne szczegóły matematyczne. Wielomian stopnia drugiego zapisujemy, jako:

$$ax^2 + bx + c$$

Wyróżnik ma postać:

$$b^2 - 4ac$$

W obliczeniach stosowany jest pierwiastek kwadratowy wyróżnika:

$$\sqrt{b^2 - 4ac}$$

jeżeli wyróżnik jest ujemny wtedy:

$$\sqrt{b^2 - 4ac} \qquad \text{znaczy} \qquad i\sqrt{-(b^2 - 4ac)}$$

gdzie i jest liczbą zespoloną , $i^2 = -1$. Rozwiązując równanie kwadratowe:

$$ax^2 + bx + c = 0$$

otrzymujemy pierwiastki równania kwadratowego, mogą to być liczby rzeczywiste lub urojone. W przypadku, gdy a = 0 i b = 0, mówimy, że mamy przypadek ekstremalnie zdegenerowany. Gdy a = 0 a b ≠ 0, mówimy o przypadku zdegenerowanym. Wtedy równanie ma postać:

$$bx + c = 0$$

i ma jeden pierwiastek:

$$x = -\frac{c}{b}$$

Listing 5.3. Równanie kwadratowe, konstrukcja if.

```
/* instrukcja if */
#include <stdio.h>
#include <math.h>        // dla sqrt()
int main()
{ float a,b,c,delta;     //wspolczynniki i wyroznik
  float x1,x2;           //pierwiastki rownania
  printf("\nax^2 +bx +c , podaj a,b,c \n");
  scanf("%f %f %f",&a,&b,&c);       //uwaga na a=0 !
  delta = b*b - 4*a*c;
  if (delta>0)
  { x1 =( -b + sqrt(delta))/(2*a);
    x2 =( -b - sqrt(delta))/(2*a);
    printf("\nx1= %f  x2= %f",x1,x2);
  }
  if (delta<0) printf("\ndelta ujemna");
  if (delta==0)
  { x1=x2=-b/(2*a);
    printf("\nx1= %f  x2= %f",x1,x2);
    printf("\nx1 = x2 = %#f",x1);    //ciekawostka
  }
  return 0;
}
```

W ogólnym przypadku ( $a \neq 0$) pierwiastki są dane, jako:

$$x_1 = \frac{1}{2a}\left(-b + \sqrt{b^2 - 4ac}\right)$$

$$x_2 = \frac{1}{2a}\left(-b - \sqrt{b^2 - 4ac}\right)$$

Jeżeli wyróżnik jest dodatni – wtedy istnieją dwa rzeczywiste pierwiastki równania. Jeżeli wyróżnik jest równy zero, wtedy dwa pierwiastki są równe i rzeczywiste. Jeżeli wyróżnik jest ujemny pierwiastki są urojone.
Poprawnie napisany program powinien uwzględniać wszystkie przypadki:

- ekstremalnie zdegenerowany
- zdegenerowany
- dwa rzeczywiste pierwiastki
- dwa jednakowe rzeczywiste pierwiastki
- dwa urojone pierwiastki

Pokazany program oblicza wyróżnik i wykonuje obliczenia, gdy wyróżnik jest większy lub równy zeru. Gdy wyróżnik jest ujemny pisze komunikat, że wyróżnik jest ujemny i nie oblicza pierwiastków. Program nie ma zabezpieczenia, gdy a = 0, co jest istotnym błędem.

Dyrektywa preprocesora:

```
#include <math.h>        // dla sqrt()
```

jest potrzebna, gdyż do obliczenia pierwiastka kwadratowego potrzebujemy funkcji bibliotecznej **sqrt().** W liniach:

```
float a,b,c,delta;         //wspolczynniki i wyroznik
float x1,x2;               //pierwiastki rownania
```

zdefiniowaliśmy zmienne, które są typu **float** (potrzebujemy liczb rzeczywistych).
Zasadnicza część programu zawarta jest w instrukcjach:

```
delta = b*b - 4*a*c;
if (delta>0)
   {   x1 =( -b + sqrt(delta))/(2*a);
       x2 =( -b - sqrt(delta))/(2*a);
       printf("\nx1= %f  x2= %f",x1,x2);
    }
```

W pierwszej linii obliczamy wyróżnik (zmienna **delta**). Następnie mamy blokową instrukcję **if**. Warunek instrukcji **if** jest obliczany i jeżeli jest prawdziwy wykonywane są instrukcje zawarte pomiędzy nawiasami klamrowymi, funkcja **printf()** drukuje rezultat obliczeń. Jeżeli warunek w instrukcji **if**:

```
if (delta>0)
```

nie jest spełniony, program omija wykonanie tej instrukcji i przechodzi do następnej:

```
if (delta<0) printf("\ndelta ujemna");
```

Jeżeli jest spełniony warunek, wykonuje się funkcja **printf()** i mamy komunikat:

```
delta ujemna
```

program przechodzi do następnej linii. Jeżeli warunek nie jest spełniony, nie wykona się funkcja **printf()** a program też przejdzie do następnej linii.

```
if (delta==0)
{ x1=x2=-b/(2*a);
  printf("\nx1= %f  x2= %f",x1,x2);
  printf("\nx1 = x2 = %#f",x1);   //ciekawostka
}
```

Instrukcja **if** (także blokowa) oblicza przypadek, gdy wyróżnik jest równy zeru. Dla celów dydaktycznych zastosowaliśmy dwa razy funkcję **printf().** Celem tego zabiegu było pokazanie jak można manipulować formatem w tej funkcji.
Zwracamy uwagę, że w tym programie, warunek w konstrukcji **if** jest sprawdzany trzy razy, bez względu, na to czy już pierwszy warunek jest spełniony (w obliczeniach nie rozważamy następnych przypadków).

Innym przykładem zastosowania instrukcji **if** program zliczający ilość wyrazów występujących w napisanym z klawiatury tekście. Program zlicza także znaki.

Listing 5.4 Zliczanie znaków i wyrazów w tekście.

```
/* instrukcja if */
//Program zlicza wyrazy i znaki
#include <stdio.h>
#include <conio.h>   //dla getche()
int main()
{ int liczba_znakow=0;
  int liczba_wyrazow=1;
  char znak;
  printf("\nPodaj tekst : \n");
  while ((znak=getche()) != '\r')
   { liczba_znakow++;
     if (znak == ' ')
      liczba_wyrazow++;
   }
  printf("\nLiczba znakow = %d",liczba_znakow);
  printf("\nLiczba wyrazow = %d", liczba_wyrazow);
  return 0;
}
```

Po uruchomieniu programu pojawia się napis:

```
Podaj tekst :
```

Jeżeli napiszemy z klawiatury tekst:

```
To jest taki sobie tekst
```

To wynikiem działania programu będzie wydruk:

```
To jest taki sobie tekst
Liczba znakow = 24
Liczba wyrazow = 5
```

Istota zliczania wyrazów polega na fakcie, że wyrazy oddzielone są spacją. Należy policzyć liczbę spacji, dodać do tej liczby 1 i otrzymujemy ilość wprowadzonych wyrazów. Tekst formalnie wprowadzany jest znak po znaku (funkcja **getche()** ). Licznik zlicza znaki.
Kluczowy fragment programu to:

```
while ((znak=getche()) != '\r')
{ liczba_znakow++;
  if (znak == ' ')
  liczba_wyrazow++;
}
```

Ponieważ wprowadzamy znaki pojedynczo musimy wielokrotnie stosować funkcje **getche().** Wielokrotne wykonywanie instrukcji nazywamy **pętlą**.

W tym przypadku użyto konstrukcji **while**. Instrukcje zamknięte pomiędzy nawiasami klamrowymi wykonywane są tak długo, dopóki wartość wyrażenia jest prawdziwa (wartość logiczna 1). W linii:

```
while ((znak=getche()) != '\r')
```

wyrażenie warunkowe ma postać:

```
((znak = getche()) !=  'r' )
```

Wprowadzany znak jest porównywany z zanegowanym znakiem **\r** (powrót karetki **CR**, uzyskujemy to po naciśnięciu klawisza Enter).
Dla przypomnienia, zapis

```
a != b          jest równoważny    ! (a == b)
```

Dopóki nie naciśniemy klawisza Enter wszystkie znaki pisane z klawiatury są kolejno przyjmowane. Enetr kończy pętlę i otrzymujemy komunikat.
Listę instrukcji pętli **while** otwiera instrukcja:

```
liczba_znakow++;
```

Jest to licznik znaków, dla przypomnienia:

```
liczba_znakow++;
```

jest równoważne

```
liczba_znakow = liczba_znakow + 1;
```

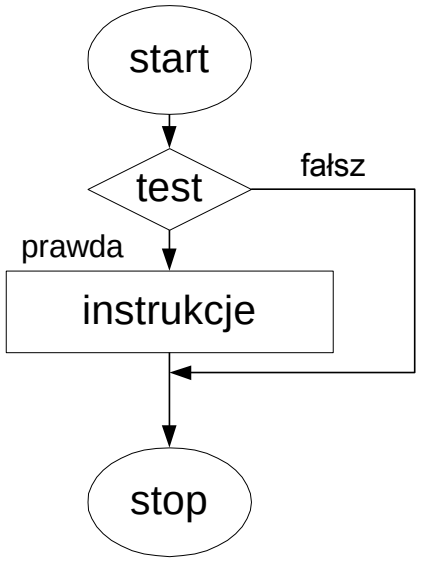
Następna instrukcja ma postać:

```
if (znak == ' ')
  liczba_wyrazow++;
```

W warunku instrukcji **if** sprawdzany jest znak spacji (puste miejsce pomiędzy dwoma apostrofami), jeżeli warunek jest spełniony licznik wyrazów **liczba_wyrazów** jest inkrementowany, w przeciwnym przypadku instrukcja ta jest pomijana. Ze względu na sposób wykrywania wyrazów, wyrazów będzie mniej niż spacji.
Dlatego w definicji zmiennych:

```
int liczba_znakow=0;
int liczba_wyrazow=1;
```

zainicjowaliśmy zmienną **liczba_wyrazow** jedynką, a nie jak dla licznika spacji zerem. Na rys. 5.1. pokazane jest działanie prostej instrukcji **if.**




Rys.51. Działanie instrukcji **if**

Instrukcje **if** mogą być zagnieżdżone – znaczy to, że w bloku pierwotnej instrukcji **if** mogą być inne instrukcje **if**. Rozważmy zapis:

```
if ( x == 1)
   if (y == 2)
     printf("\n+++");
```

Te trzy linie możemy rozpatrywać jako zapis:

```
if (x == 1)  instrukcja;
```

gdzie **instrukcja** ma postać:

```
if (y == 2)  printf("\n+++");
```

Drugi warunek **if** testuje się tylko wówczas, gdy prawdziwy jest warunek zawarty w pierwszym **if**. Wobec tego funkcja **printf()** będzie wywołana tylko wtedy, gdy oba warunki są spełnione jednocześnie. Należy pamiętać, że dwie zagnieżdżone instrukcje **if** można zastąpić jedną instrukcją z operatorem logicznym **&&** .

Innym przykładem zastosowania konstrukcji wielokrotnego **if** jest program pokazany na listingu 5.5. W programie sprawdzamy, czy liczba jest większa/równa liczbie 100 czy jest mniejsza. Jeżeli liczba jest mniejsza i ujemna to chcemy otrzymać także tą informację. Dana liczba jest wprowadzana z klawiatury, przy pomocy złożonej konstrukcji **if** sprawdzamy wprowadzoną liczbę.

Listing 5.5 Zagnieżdżona instrukcja if

```
/* zagniezdzone if */
#include <stdio.h>
int main()
{ const int granica = 100;
  int liczba;
  printf("\npodaj liczbe calkowita : ");
  scanf("%d",&liczba);
  if (liczba < granica)
   { printf("liczba jest mniejsza od %d",granica);
     if (liczba < 0)
       printf(" i liczba jest ujemna");
   }
  if (liczba >= granica)
  printf("liczba jest rowna lub wieksza od %d",
          granica);
  return 0;
}
```

Działanie programu jest następujące:

```
podaj liczbe calkowita : 0
liczba jest mniejsza od 100
podaj liczbe calkowita : -150
liczba jest mniejsza od 100 i liczba jest ujemna
podaj liczbe calkowita : 100
liczba jest rowna lub wieksza od 100
```

W następującym fragmencie programu zastosowaliśmy zagnieżdżoną konstrukcję **if**.

```
if (liczba < granica)
{ printf("liczba jest mniejsza od %d",granica);
  if (liczba < 0)
    printf(" i liczba jest ujemna");
}
```

W programie zwracamy uwagę na sposób inicjalizacji zmiennej **granica**:

```
const int granica = 100;
```

W tej linii zadeklarowaliśmy stałą typu **int** o wartości 100.

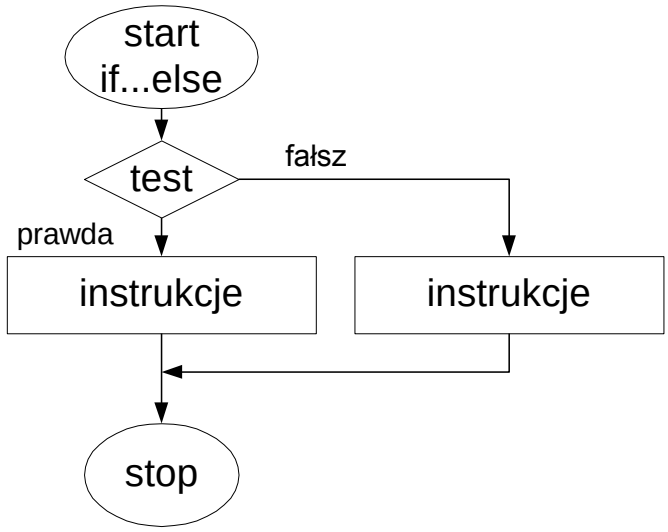
## 5.5. Instrukcja if – else

Konstrukcja **if** pozwala na wykonanie instrukcji, gdy warunek jest prawdziwy. Wygodnie jest dysponować konstrukcją, która pozwoli wykonać jedną instrukcję, gdy warunek jest prawdziwy lub wykonać inną instrukcje, gdy warunek jest fałszywy. W języku C dysponujemy taką konstrukcją. Jest to konstrukcja **if – else**.

Konstrukcja **if – else** ma postać:

```
if (wyrażenie)
  instrukcja_1
else
  instrukcja_2
instrukcja _nastepna
```

Jeżeli wyrażenie jest prawdziwe, wtedy wykonuje się **instrukcja_1** a **instrukcja_2** jest ominięta. Jeżeli wyrażenie jest fałszywe, wtedy **instrukcja_1** jest ominięta i wykonuje się **instrukcja_2.** W obu przypadkach wykonana jest **instrukcja_następna**. Rysunek 5.2. pokazuje działanie konstrukcji **if…else.**




Rys.5.2 Zasada działania konstrukcji **if ...else**

Rozważmy zapis:

```
if (a < b)
   min = a
else
   min = b
```

Jeżeli warunek **a<b** jest prawdziwy, wtedy zmiennej **min** będzie przypisana wartość **a** jeżeli ten warunek jest fałszywy wtedy zmiennej **min** będzie przypisana wartość **b.**

Użycie konstrukcji **if – else** zademonstrowane jest w kolejnym programie (listing 5.6). Program oblicza podatek, jaki mamy zapłacić od pensji. Założono, że jeżeli pensja jest mniejsza od 3000 zł (zmienna **granica**) to wtedy podatek wynosi 20 % (zmienna **pr_1**). Jeżeli pensja jest większa lub równa 3000 zł wtedy podatek wynosi 45 % (zmienna **pr_2**).

Listing 5.6 Konstrukcja if…else.

```
/* instrukcja if-else */
#include <stdio.h>
#include <conio.h>
int main()
{ float pensja;
  float granica = 3000.0;
  float pr_1 = 0.20;
  float pr_2 = 0.45;
  printf("\nprogram oblicza podatek od pensji\n");
  printf("\n Podaj swoja pensje : ");
  scanf("%f",&pensja);
  if (pensja < granica)
    printf("\npodatek = %8.2f zl",pensja*pr_1);
  else
    printf("\npodatek = %8.2f zl",pensja*pr_2);
  return 0;
}
```

Po uruchomieniu programu mamy wydruk:

```
program oblicza podatek od pensji
Podaj swoja pensje :  1000.0
podatek =  200.00 zl
```

W programie zmienne takie jak **granica**, **pr_1** i **pr_2** zostały zadeklarowane, jako zmienne typu **float**. Obliczenia są wykonywane na liczbach rzeczywistych wobec tego taki typ jest wymagany. Zasadnicza część programu jest zawarta w następującym fragmencie:

```
scanf("%f",&pensja);
if (pensja < granica)
   printf("\npodatek = %8.2f zl",pensja*pr_1);
else
   printf("\npodatek = %8.2f zl",pensja*pr_2);
```

W instrukcji **if...else** sprawdzany jest warunek i w zależności od wyniku podejmowana jest akcja: albo obliczany jest podatek od pensji w wysokości 20 procent albo 45 procent. Należy zwrócić uwagę na funkcję **printf**():

```
printf("\npodatek = %8.2f zl",pensja*pr_1);
```

Wzór na obliczenie podatku:

```
pensja*pr_1
```

zapisany jest bezpośrednio w części programu, gdzie zwykle umieszcza się zmienne. Dzięki temu zapisowi nie musieliśmy definiować jeszcze jednej zmiennej, takiej jak np.

```
podatek_1 = pensja*pr_1
```

aby umieścić ją bezpośrednio w instrukcji z funkcją **printf().** Konstrukcja **if...else** może być użyta jako część konstrukcji **if:**

```
if (x == 1)
  if (y == 2)
    printf("**");
  else
    printf("++++");
```

Możemy zapytać się czy **else** związane jest z pierwszym czy też z drugim **if**. Generalnie obowiązuje zasada:

***else** związane jest z najbliższym **if***

W podanym przykładzie mamy konstrukcję:

```
if (x == 1)
   instrukcja
```

gdzie **instrukcja** oznacza:

```
if (y == 2)
   printf("**");
else
   printf("++++");
```

Możemy budować skomplikowane instrukcje składające się z wielu konstrukcji **if...else.** Przykładem może być program pokazany na listingu 5.7. Program zlicza liczbę wystąpień liczb, liter, spacji oraz innych znaków występujących w tekście wprowadzonym z klawiatury. Do wprowadzenia tekstu wykorzystaliśmy funkcje **getche()**. Zmienne przechowujące liczbę wystąpień znaków są zadeklarowane i zainicjowane w następujący sposób:

```
int ile_spac,ile_liczb,ile_liter,ile_innych;
ile_spac = ile_liczb = ile_liter = ile_innych = 0;
```

Zasadnicza część programu ma postać:

```
while ((znak = getche()) != '\r')
{ if (znak == ' ')
    ++ile_spac;
  else if ('0' <= znak && znak <= '9')
    ++ile_liczb;
  else if ('a' <= znak && znak <= 'z' ||
           'A' <= znak && znak <= 'Z')
    ++ile_liter;
  else ++ile_innych;
}
```

Listing 5.7 Zliczanie wystąpień znaków

```
/* dzialanie if else */
#include <stdio.h>
#include <conio.h>
int main()
{ char znak;
  int ile_spac,ile_liczb,ile_liter,ile_innych;
  ile_spac = ile_liczb = ile_liter = ile_innych = 0;
  printf("\n......Wpisz tekst ..........\n");
  while ((znak = getche()) != '\r')
  { if (znak == ' ')
      ++ile_spac;
    else if ('0' <= znak && znak <= '9')
      ++ile_liczb;
    else if ('a' <= znak && znak <= 'z' ||
             'A' <= znak && znak <= 'Z')
      ++ile_liter;
    else ++ile_innych;
  }
  printf("\n\n%10s%10s%10s%14s",
         "spacje", "liczby","litery","inne znaki");
  printf("\n\n%10d%10d%10d%10d\n\n",
          ile_spac,ile_liczb,ile_liter,ile_innych);
  return 0;
}
```

Po uruchomieniu program działa w następujący sposób.

```
......Wpisz tekst ..........
To jest numer telefonu (+81) 133-1333
Spacje liczby   litery inne znaki
        5        9                  19          4
```

Pętla **while** pozwala na kolejne wczytywanie znaków tekstu. Za każdym razem sprawdzany jest warunek:

```
((znak = getche()) != '\r')
```

Zliczanie wystąpień spacji ma postać:

```
if (znak == ' ')
  ++ile_spac;
```

W konstrukcji **if** sprawdzany jest warunek, jeżeli jest spełniony (wprowadzony znak jest znakiem spacji) wtedy wykonywana jest instrukcja inkrementacji. Bardziej skomplikowany jest sposób zliczania wystąpień liter, ponieważ mogą być litery duże i małe:

```
else if ( 'a' <= znak && znak <= 'z' ||
          'A' <= znak && znak <= 'Z')
++ile_liter;
```

Wyrażenie w konstrukcji **if** ma postać:

```
('a'<=znak && znak<='z' ||  'A'<=znak && znak<='Z')
```

Zastosowano dwa operatory logiczne:

- && - logiczne AND
- || - logiczne OR

Wyrażenia:

```
'a' <= znak && znak <= 'z'
```

oraz

```
'A' <= znak && znak <= 'Z')
```

sprawdzają czy znak jest małą literą czy dużą. Pamiętamy, że kody znaków ASCII dla dużych liter, zawarte są od 65 (A) do 90 (Z), a kody dziesiętnie dla małych liter od 97 (a) do 122 (z). ). Określamy, czy znak należy do przedziału (65,90) czy (97,122). Znak jest literą, jeżeli należy do przedziału pierwszego lub drugiego. Do sprawdzenia tego faktu wykorzystany jest operator logiczny **OR** (symbol ||) . Jeżeli ten warunek jest spełniony następuje inkrementacja licznika:

```
++ile_liter;
```

Wydruk wyniku zapewniają instrukcje:

```
printf("\n\n%10s%10s%10s%14s",
          "spacje", "liczby","litery","inne znaki");
   printf("\n\n%10d%10d%10d%10d\n\n",
           ile_spac,ile_liczb,ile_liter,ile_innych);
```

W pierwszej instrukcji funkcja **printf()** wypisuje łańcuch, dlatego użyto specyfikacji **%10s**, w drugiej instrukcji drukowany jest wynik. Jeżeli instrukcja nie mieści się w jednej linii, lub chcemy ją z jakichś powodów przenieść do następnej linii – czynimy to bez kłopotu. Kompilator szuka znaku średnika ” ; ” , co oznacza koniec instrukcji. Zwróćmy uwagę na sposób zapisu instrukcji z konstrukcją **if – else**. Formalnie mamy zapis:

```
if (wyrażenie1)
   instrukcja1
else if (wyrażenie2)
   instrukcja2
else if (wyrażenie3)
   instrukcja3
.....................
else if (wyrażenieN)
   instrukcjaN
instrukcja_nastepna
```

Jest to oczywiście konstrukcja wielokrotna **if – else**, ale zapisana w bardziej chyba czytelniejszy sposób. Czasami taki zapis nosi nazwę ***konstrukcji else – if*** .

Klasycznym przykładem zastosowania konstrukcji **else - if** może być program wykonujący cztery działania (dodawanie, odejmowanie, dzielenie i mnożenie). Postać programu pokazana jest na kolejnym listingu. Jak widzimy, użytkownik, zgodnie z komunikatem musi wpisać działanie, np.:

```
2.0 * 2.0
```

program obliczy iloczyn i poda wynik. Funkcja **printf()** do umieszczenia na ekranie wyniku wykorzystuje specyfikator **%.2e** .

```
printf("\nwynik = %.2e",wynik);
```

Jest to zapis liczby zmiennoprzecinkowej w ***notacji wykładniczej***, w naszym przypadku, z dwoma miejscami po kropce dziesiętnej. Przykładem tej notacji jest zapis:

```
15.00e+02
```

Listing 5.8 Kalkulator, konstrukcja else…if

```
/* else-if; kalkulator */
#include <stdio.h>
int main()
{ char oper;
  float liczba1=1.0, liczba2=1.0;
  float wynik = 0.0;
  while (!(liczba2==0.0))
  { printf("\npodaj: liczbe operator liczbe,
             np. 2.0 + 2.0\n");
    scanf("%f %c %f",&liczba1,&oper,&liczba2);
    if      (oper == '+') wynik = liczba1 + liczba2;
    else if (oper == '-') wynik = liczba1 - liczba2;
    else if (oper == '*') wynik = liczba1*liczba2;
    else if (oper == '/') wynik = liczba1/liczba2;
    printf("\nwynik = %.2e",wynik);
  }
return 0;
}
```

Aby zakończyć program musimy wprowadzić dane w postaci:

```
0.0  0.0
```

ponieważ czytana jest liczba typu **float** za pomocą funkcji **scanf()**. Warunek sprawdzany jest w konstrukcji **while**:

```
while (!(liczba2==0.0))
```

W wyrażeniu warunkowym wykorzystano unarny operator negacji (symbol ! ) . Wyrażenie to zapobiega także, wykonaniu dzielenia przez zero.

## 5.6. Instrukcje switch, break i continue

Zagnieżdżone konstrukcje **if...else** są często bardzo nieczytelne. Dlatego w języku C wprowadzono konstrukcję **switch**, która jest uogólnieniem (alternatywą) dla złożonych konstrukcji **if...else**.
Zanim jednak omówimy konstrukcję **switch**, opiszemy specjalną instrukcję **break** (drugą pożyteczną instrukcję **continue** omówimy przy opisie pętli), ponieważ konstrukcja **switch** bardzo często wykorzystuje ją. Instrukcje **break** i **continue** kontrolują sekwencje wykonywania instrukcji.

### Instrukcja break

Instrukcja **break** powoduje wyjście z wnętrza pętli lub instrukcji **switch**. Stosowana jest w przypadkach, gdy zajdzie nieoczekiwany warunek lub specjalna sytuacja (np., gdy nie ma potrzeby wykonywania kolejnych instrukcji pętli). Następujący program (listing 5.9) prezentuje użycie instrukcji **break**.

Listing 5.9 Instrukcja break

```
/* instrukcja break */
#include <stdio.h>
#include <math.h>  //dla funkcji sqrt()
int main()
{ float x, bok_kwadrat;
  while (1)
  { printf("\nPodaj pole kwadratu : ");
    scanf("%f",&x);
    if (x<0.0)
      break;
    bok_kwadrat = sqrt(x);
    printf("\n%.2f",bok_kwadrat);
  }
  printf("\nObliczono bok kwadratu");
  return 0;
}
```

W programie pokazano użycie instrukcji **break**. Należy zwrócić uwagę na zapis:

```
while(1)
```

który reprezentuje tzw. ***pętlę nieskończoną.*** W pokazanej konstrukcji, warunek jest testowany w instrukcji **if** i po spełnieniu zakładanego warunku, nieskończona pętla jest przerywana. W programie włączony jest plik biblioteczny **<math.h>** do obsługi funkcji **sqrt()**. Funkcja **sqrt()** oblicza pierwiastek kwadratowy liczby, jeżeli liczba jest ujemna, nie chcemy dalej kontynuować obliczeń. W programie, jeżeli zmienna **x** jest ujemna, wywoływana jest instrukcja **break**, następuje opuszczenie blokowej instrukcji **while (1) {.....}** i sterowanie przekazywane jest do instrukcji:

```
printf("\nObliczono bok kwadratu");
```

umieszczonej tuż przed końcem programu. Kolejny program (listing 5.10) do zgadywania liczb, demonstruje użycie instrukcji **break** a także stosuje szybką metodę przeszukiwania opartą na technice podziału określonego przedziału. Program odgaduje liczbę z przedziału (1,99) pomyślaną przez użytkownika. Dialog z programem wygląda następująco (wybrana przez nas liczba to 37):

```
Pomysl liczbe z przedzialu 1,99
nacisnij jeden z klawiszy <,=,> w zaleznosci od
pytania
Liczba rowna,wieksza,mniejsza od 50  <
Liczba rowna,wieksza,mniejsza od 25  >
Liczba rowna,wieksza,mniejsza od 38  <
Liczba rowna,wieksza,mniejsza od 31  >
Liczba rowna,wieksza,mniejsza od 34  >
Liczba rowna,wieksza,mniejsza od 36  >

Liczba pomyslana to 37
```

Listing 5.10 Szybkie przeszukiwanie, instrukcja break

```
/* instrukcja break */
#include <stdio.h>
#include <conio.h>
int main()
{ float liczba, sk;
  char znak;
  printf("\nPomysl liczbe z przedzialu 1,99");
  printf("\nnacisnij jeden z klawiszy <,=,> w
          zaleznosci od pytania");
  liczba = sk = 50.0;
  while (sk > 1.0)
  { printf("\nLiczba rowna,wieksza,mniejsza od %.0f ",
            liczba);
    sk /= 2.0;
    if ( (znak = getche()) == 0x3D)
      break;
    else if (znak == 0x3E)
      liczba += sk;
    else
      liczba -= sk;
  }
  printf("\nLiczba pomyslana to %.0f ",liczba);
  return 0;
}
```

W deklaracji zmiennych przypisano wartości początkowe:

```
liczba = sk = 50.0;
```

Poszukiwanie pomyślanej liczby program zaczyna od ustalenia wartości równej połowie przedziału (zmienna **sk**).

W kolejnych krokach dzielony jest przedział na połowy:

```
sk /= 2.0
```

i w zależności od odpowiedzi obliczana jest proponowana zmienna **liczba**:

```
else if (znak == 0x3E)
  liczba += sk;          // liczba = liczba + sk
else
  liczba -= sk;          // liczba = liczba - sk
```

a na końcu produkowane jest zapytanie, czy jest to wartość pomyślana. W wyrażeniu warunkowym:

```
if (znak == 0x3E)
```

zapis 3E oznacza kod ASCII znaku > w zapisie szesnastkowym ( znak = ma kod 3D, znak < ma kod 3C ). W języku C liczbę w zapisie szesnastkowym należy poprzedzić symbolem **0x** lub **0X** (zero, iks). Możemy w tym miejscu alternatywnie zastosować zapis dziesiętny lub ósemkowy. Oczywiście może się zdarzyć, że zaproponowana liczba jest równa pomyślanej, wtedy program należy przerwać. Zabezpiecza to następujący fragment programu stosujący instrukcje **break**:

```
if ( (znak = getche()) == 0x3D)
break;
```

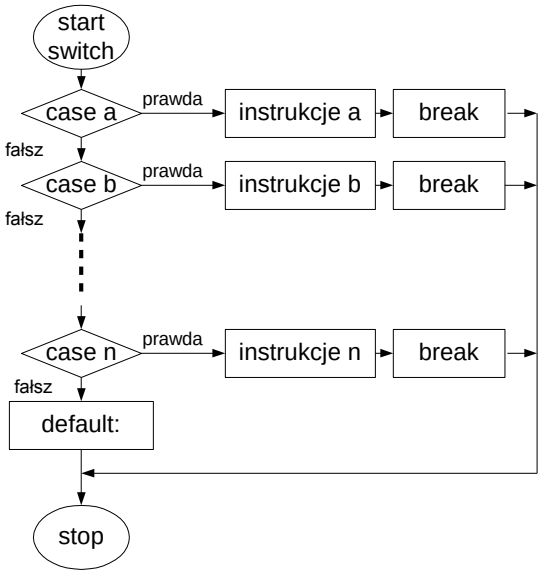
W pętli **while** program testuje czy zmienna **sk** osiągnęła wartość 1.0.
Jeżeli mamy wiele wariantów, zagnieżdżone konstrukcje **if...else** stają się mało czytelne i często są źródłem błędów. W języku C do obsługi sytuacji wielowariantowych służy instrukcja **switch**. W porównaniu do konstrukcji **if...else** ma bardziej prostą budowę i jest bardziej elastyczna.
Syntaktycznie instrukcja **swich** ma postać:

```
switch (wyrażenie)
{
  case wyrażenie_stałe_1 :
     instrukcje_1;
  case wyrażenie_stałe_2 :
     instrukcje_2;
    ......................
  case wyrażenie_stałe_n :
     instrukcje_n;
  default :
     inne_instrukcje
}
```

Argumentem instrukcji **switch** jest zmienna typu **char** lub **int** (całkowita). W nawiasach klamrowych umieszczone są polecenia **case** i **default**. Z każdym poleceniem **case** jest związane wyrażenie stałe.
Gdy wartość wyrażenia w instrukcji **switch** pasuje do wyrażenia stałego związanego z konkretnym poleceniem **case**, wykonywane są instrukcje

następujące po tym poleceniu. Wyrażenia stałe związane z poleceniem **case** muszą być jednoznaczne i unikalne. Grupę instrukcji związanych z konkretnym polecenie **case** zazwyczaj kończy instrukcja **break**, jeżeli nie będzie tej instrukcji, wykona się następne polecenie **case**. Instrukcja **break** powoduje natychmiastowe opuszczenie konstrukcji **switch**.
Działanie konstrukcji **switch** (rys.5.3) jest następujące:

1 obliczane jest wyrażenie w instrukcji **switch**
2 wykonywane jest polecenie **case**, gdy wyrażenie stałe związane z **case** ma wartość wyrażenia **switch**, jeżeli nie znajdzie się takiej zgodności to wykonywane jest polecenie **default** i kończy się blok **switch** (polecenie **default** nie jest konieczne).
3 blok **switch** jest kończony po napotkaniu pierwszego polecenia **break**.




Rys.5.3.Struktury wielokrotnego wyboru **switch**

Przykładowy program wykorzystujący instrukcję **switch** pokazany jest na listingu 5.11. Program symuluje działania prostego kalkulatora, można wykonywać cztery podstawowe działania arytmetyczne.

Listing 5.11 Kalkulator, konstrukcja switch

```
/* instrukcja switch - kalkulator */
#include <stdio.h>
int main()
{ char oper;
  float liczba1, liczba2;
  printf("\npodaj liczbe, operator, liczbe\n");
  scanf("%f %c %f",&liczba1,&oper,&liczba2);
  switch(oper)
  {
   case '+':
    printf("\nSuma    = %f",liczba1 + liczba2);
    break;
   case '-':
    printf("\nRoznica = %f",liczba1 - liczba2);
    break;
   case '*':
    printf("\nIloczyn = %f",liczba1 * liczba2);
    break;
   case '/':
    if (liczba2==0.0)
    {
      printf("\nNie dziel przez zero");
      break;
    }
    else
    printf("\nIloraz = %f",liczba1 / liczba2);
    break;
    default :
      printf("\nNieakceptowany operator");
  }
  return 0;
}
```

Instrukcja **switch** jest bardzo wygodna do programowania np. struktury menu.

## 5.7. Operator warunkowy i konstrukcja if…else

Operator warunkowy jest jednym z bardziej nietypowych operatorów występujących w języku C. Przypomnijmy, ten operator zbudowany jest z dwóch symboli znaku zapytania **?** i dwukropka **:** , do działania potrzebuje trzech wyrażeń. Pozwala na zwarty zapis instrukcji wymagających użycia instrukcji **if.** Zapis instrukcji z operatorem warunkowym ma postać:

```
wyrażenie_warunkowe ? wyrażenie_1 : wyrażenie_2 ;
```

W tego typu konstrukcji jest obliczane **wyrażenie_warunkowe**. Jeżeli jest prawdziwe, to wtedy wyliczane jest **wyrażenie_1** a wartość tego wyrażenia jest przypisywana do wyrażenia warunkowego. Jeżeli **wyrażenie_warunkowe** jest

fałszywe to wyliczana jest **wyrażenie_2,** a jego wartość jest przypisywana do wyrażenia warunkowego. Zwykle wyrażenie z operatorem warunkowym jest równoważne konstrukcji **if...else**. Rozważmy przykład wyrażenia z operatorem warunkowym:

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

Jeżeli liczba **a** jest większa od liczby **b** to zmiennej **maks** przypisana jest wartość zmiennej **a,** jeżeli liczba **a** jest mniejsza od **b**, to wtedy zmiennej **maks** przypisana jest wartość zmiennej **b**. Równoważny zapis powyższej instrukcji jest następujący:

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

Na listingu 5.12 pokazano przykładowe wykorzystanie operatora warunkowego. Program wczytuje ilość bitów, które musimy ustawić, program wylicza ile bajtów potrzebujemy, aby ustawić żądaną ilość bitów. Zakładamy, że jeden bajt to osiem bitów.

Listing 5.12 operator warunkowy :?

```
#include <stdio.h>
#include <conio.h>
#define blok 8    //1 bajt to 8 bitow
int main()
{ int bity, bajty;
  puts("ile bitow zapisac?\n");
  while (scanf("%d", & bity) == 1)
  { bajty = bity/blok;
    bajty += ((bity % blok == 0 )) ? 0 : 1;
    printf("ustawiamy %d %s \n", bajty,
              bajty == 1 ? "bajt" : "bajty");
    puts("ile bitow zapisac, (q - konczy):\n");
  }
  getche();
  return 0;
}
```

Wynikiem działania programu z listingu 5.12 jest komunikat:

```
ile bitow zapisac?
6
ustawiamy 1 bajt
ile bitow zapisac, (q konczy):
9
ustawiamy 2 bajty
ile bitow zapisac, (q konczy):
q
```

# ROZDZIAŁ 6

## INSTRUKCJE POWTARZANIA

## 6.1. Wstęp

Opracowując programy, potrzebujemy mechanizmów umożliwiających wielokrotne powtarzanie listy instrukcji. W językach programowania stosowany jest mechanizm powtarzania zwany pętlami. Język C oferuje trzy takie konstrukcje:

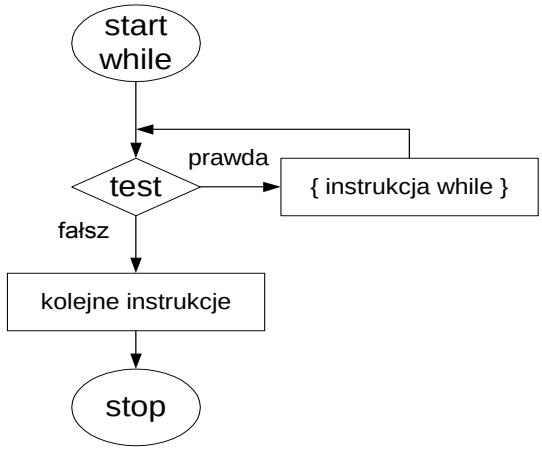
- pętla **while**
- pętla **do...while**
- pętla **for**

## 6.2. Instrukcja while

Pętlę **while** stosujemy zazwyczaj, gdy nie znamy liczby powtórzeń. Warunek wykonania pętli testowany jest przed każdym, nawet pierwszym wykonaniem pętli. Gdy warunek nie jest spełnionym pętla nie będzie wykonana ani razu. Struktura pętli **while** jest następująca:

```
while (wyrażenie)
   instrukcja_1;

lub
while (wyrażenie)
 {
   instrukcja_1;
   instrukcja_2;
   ..................
   instrukcja_n;
 }
```

Budowa pętli **while** pokazana jest na rys.6.1.




Rys.6.1 Budowa pętli **while**.

Przykłady użycia pętli **while** mogą być następujące:

```
while (i++ < n )
   liczba *= i;

while ((znak = getchar())  != EOF)
{
   if ('a' <= znak && znak <= 'z')
   ++liczba_malych _liter;
   ++liczba_wszystkich;
}
```

Instrukcje w pętli **while** będą wykonywane dopóki wyrażenie będzie spełnione (wartość wyrażenia będzie różna od zera). Aczkolwiek pętla **while** jest najprostszą pętlą, należy uważać przy kodowaniu, aby nie wystąpiła sytuacja, gdy pętla **while** będzie nieskończona (chyba, że tak zadecydujemy). Program służący do obliczania kwadratów liczb całkowitych od 1 do 10 pokazujący zastosowanie instrukcji **while** przedstawiony jest na listingu 6.1.

Listing 6.1 Instrukcja while

```
#include <stdio.h>
int main()
{int liczba, kwadrat;
 liczba = 1;
 while (liczba < 11)
 {
  kwadrat = liczba*liczba;
  printf("\nkwadrat liczby %3d=%4d",liczba,kwadrat);
  liczba = liczba + 1;
 }
  return 0;
}
```

Zmiennej **liczba,** która jest typu **int**, przypisana jest wartość 1. Pętla **while** musi się wykonać, ponieważ warunek:

```
liczba < 11
```

jest spełniony, wartość zmiennej **liczba** na starcie jest mniejsza niż 11. Gdyby np. wartość zmiennej **liczba** na początku była równa 50, warunek nie byłby spełniony, pętla nie wykonałaby się. Ponieważ na starcie zmienna **liczba** ma wartość 1, instrukcje pętli **while**, znajdujące się w nawiasach klamrowych wykonają się. Obliczony zostanie kwadrat liczby i wydrukowany. W instrukcji:

```
liczba = liczba + 1;
```

zmiennej **liczba** przypisana jest nowa wartość (teraz zmienna liczba ma wartość 2). Program doszedł do końca pętli **while**, ponownie sprawdza warunek. Ponieważ warunek jest prawdziwy, instrukcje w nawiasach klamrowych znowu się wykonają. Pętla **while** będzie powtarzała wykonanie tych instrukcji, dopóki

warunek nie będzie fałszywy. Nastąpi to, gdy zmienna **liczba** przyjmie wartość 11. Wtedy program przejdzie do następnej instrukcji, i zakończy działanie. Omawiany program nie jest oszczędny. Poniżej prezentujemy bardziej zwartą postać.

Listing 6.2 Instrukcja while

```
#include <stdio.h>
int main()
{ int liczba = 0;
  while (liczba++ < 10)
  printf("\nx = %2d x*x = %4d",liczba,liczba*liczba);
  return 0;
}
```

W wyrażeniu warunkowym instrukcji **while**:

```
while (liczba++ < 10)
```

jest sprawdzany warunek i następuje inkrementacja zmiennej **liczba**. Zmienna **liczba** musi być zainicjowana wartością 0, ponieważ bezpośrednio po sprawdzeniu warunku, jej wartość jest zwiększana o 1. Z tego samego powodu, wartość graniczna jest równa 10.
W kolejnym programie zastosujemy konstrukcję **while** aby wydrukować tekst na ekranie dużymi literami, gdy z klawiatury wprowadzimy tekst, pisany małymi literami. W programie wykorzystano funkcję biblioteczną **toupper(),** dzięki której małe litery zamieniane są na duże.

Listing 6.3 .Drukowanie dużych liter, instrukcja while

```
#include <stdio.h>
#include <ctype.h> //dla toupper
#define EOL '\n'
#define DL 50

int main()
{ int ilosc;
  int k = 0;
  char tekst[DL];
  printf("\nwprowadz tekst z klawiatury\n");
  while ((tekst[k++]=getchar()) != EOL )
            if (k == DL) break;
  printf("\nilosc znakow = %d\n",ilosc = k);
  k = -1;
  while (++k < ilosc)
            putchar(toupper(tekst[k]));
  return 0;
}
```

Do wczytywania znaków z klawiatury zastosowano funkcję **getchar().**

Wprowadzane znaki nasz program traktuje, jako elementy tablicy:

```
tekst[k++]=getchar()
```

W tym zapisie indeks tablicy jest inkrementowany. Tablica jest zadeklarowana w następujący sposób:

```
char tekst[DL];
```

Rozmiar tablicy został ustalony w instrukcji preprocesora:

```
#define DL 50
```

Zadeklarowano tablicę o 50 elementach. Wczytywanie znaków realizuje instrukcja **while**:

```
while ((tekst[k++]=getchar()) != EOL )
    if (k == DL) break;
```

Instrukcja **break** jest zastosowana, aby kontrolować ilość wypisywanych znaków. Znaki są wczytywane dopóki nie są równe stałej **EOL**. Stała ta jest zdefiniowana w dyrektywie preprocesora:

```
#define EOL '\n'
```

Wydruk liter obsługują instrukcje:

```
k = -1;
while (++k < ilosc)
   putchar( toupper( tekst [k] ) ) ;
```

Kolejnym przykładem zastosowania pętli **while** jest program obliczający iloczyn skalarny dwóch wektorów. Wektory maja postać:

$$\mathbf{X} = (x_1, x_2, \ldots x_n)$$
$$\mathbf{Y} = (y_1, y_2, \ldots y_n)$$

Iloczyn skalarny:

$$\mathbf{S = X \bullet Y}$$

obliczamy następująco:

$$S = \sum_{i=1}^{n} x_i * y_i$$

Do obliczeń musimy podać wymiar wektora oraz składowe wektorów x i y. Dane wprowadzamy parami $x_1, y_1, x_2, y_2$ ....., $x_n, y_n$ .

Listing 6.4 Iloczyn skalarny, instrukcja while

```
#include <stdio.h>
#include <conio.h>      //DOS, kbhit()
int main()
{ int n;
  float x,y;
  int i = 0;
  float s = 0.0;
  printf("\nPodaj wymiar : ");
  scanf("%d",&n);
   while (i++ < n)
     {printf("skladowa %d :  ",i);
      scanf("%f%f",&x,&y);
      s += x*y;
     }
   printf("iloczyn skalarny = %.2f",s);
   while (!kbhit());     //przytrzymanie ekranu
  return 0;
}
```

W pętli **while** czytane są składowe wektora i obliczany jest iloczyn skalarny. Ten kod oszczędza wyrażenie, ponieważ inkrementacja jest wykonywana bezpośrednio w warunku **while** przed jego sprawdzeniem:

```
while (i++ < n)
```

Jako interesujący przykład zastosowania pętli **while**, należy traktować następujący program, który kopiuje pamięć ekranu.

Listing 6.4 Kopiowanie pamięci ekranu, instrukcja while

```
/* pamiec ekranu no1 */
#include <stdio.h>
#include <conio.h>
#define ROZMIAR 2000
int main()
{int far *wsk;
 int miejsce;
 char znak;
 printf("\nekran pokrywa sie znakami,wprowadzaj rozne
         znaki");
 printf("\nEnter konczy program,napisz znak aby
         wystartowac: ");
 wsk = (int far *) 0xB8000000L;
 while( (znak=getch()) != '\r')
   for(miejsce=0; miejsce<ROZMIAR; miejsce++)
        *(wsk + miejsce) = znak | 0x0700;
 return 0;
}
```

Wyrażenie sterujące działaniem pętli **while** może być bardzo efektywne. Bardzo często możemy stosować następująca konstrukcję:

```
suma = 0;
while (scanf(„%d”, &liczba) == 1)
   suma = suma + liczba;
```

W tym fragmencie kodu wczytujemy kolejne liczby, aby je sumować. Pętla **while** powtarza operacje czytania liczby z klawiatury tak długo, dopóki wartość zwracana przez funkcje **scanf()** jest 1. Taka wartość jest zwracana, gdy funkcja **scanf()** otrzymuje dane zgodne z zadeklarowanym formatem, w naszym przypadku, gdy wprowadzane są liczby całkowite. W momencie wprowadzenia dowolnego znaku (np. a, b, q, itp.), funkcja **scanf()** zwróci wartość 0 i działanie pętli **while** będzie zakończone. Pamiętamy, że wywołanie funkcji **scanf()** umieszcza odczytaną wartość w zmiennej **liczba**. Jest to bardzo oszczędne kodowanie, pętla **while** jest pętlą nieskończoną, ale z dobrze określonym warunkiem wyjścia, jednocześnie w wyrażeniu warunkowym wczytujemy potrzebne dane. Kompletny program wykorzystujący tego typu konstrukcje pokazany jest na listingu 6.5. Program odczytuje wprowadzone z klawiatury liczby całkowite w pętli **while**:

```
while (scanf("%d",&liczba) == 1)
 { KOM;
   suma = suma + liczba;
 }
```

i oblicza sumę. W momencie wprowadzenia dowolnego znaku z klawiatury, program kończy czytanie liczb i podaje sumę końcową.

Listing 6.5 Instrukcja while,

```
#include <stdio.h>
#include <conio.h>
#define KOM printf("\n podaj liczbe : ")
int main()
{int suma, liczba;
 suma = 0;
 printf("\n dowolny znak konczy program");
 KOM;
 while (scanf("%d",&liczba) == 1)
  { KOM;
     suma = suma + liczba;
  }
 printf("\n suma = %d", suma);
 getche();
 return 0;
}
```

## 6.3. Instrukcja do...while

Działanie tej pętli jest bardzo zbliżone do działania pętli **while**. Pętla **do...while** jest stosowana wtedy, gdy nie znamy dokładnie liczby powtórzeń, ale chcemy, aby instrukcje leżące w jej zasięgu były wykonane przynajmniej jeden raz. Pamiętamy, że pętla **while** sprawdza najpierw wyrażenie warunkowe i w zależności od wartości tego wyrażenia wykona się lub nie. Pętla **do...while** najpierw się wykonuje a potem sprawdza wyrażenie warunkowe.
Postać pętli **do...while** jest następująca:

```
do
{
   instrukcja1;
   instrukcja2;
   ..................
   instrukcjaN;
}  while (wyrażenie);
```

Instrukcje zamknięte w nawiasie klamrowym będą wykonywane tak długo, dopóki będzie spełniony warunek w wyrażeniu konstrukcji **do...while**. Pętla **do…while** zostaje zawsze wykonana przynajmniej jeden raz. Pętla **for** i **while** mogą natomiast nie zostać wykonane ani razu. Rekomenduje się stosowanie pętli **do…while**, gdy na pewno jest wymagana przynajmniej jedna iteracja. Należy unikać stosowania pętli **do…while**, gdy zestaw instrukcji jest powtarzany na wyraźne polecenie użytkownika. Typowym przykładem jest pytanie ukazujące się na ekranie monitora:

```
   czy liczyć dalej  (T lub N) ?
```

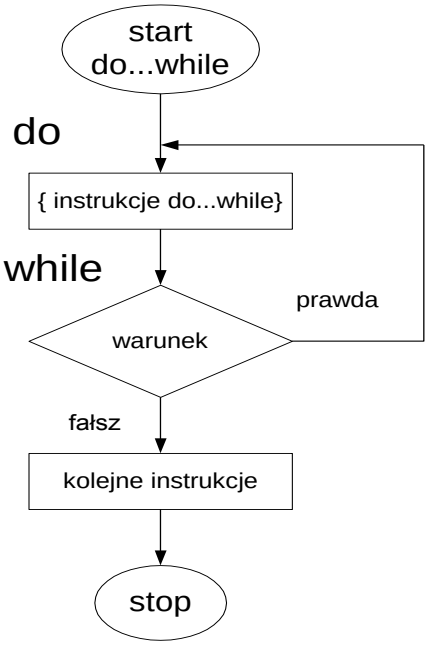
Jeżeli iteracje obsługuje pętla **do…while**, to, mimo, że użytkownik odpowie N (nie), działania zostaną wykonane, ponieważ test następuje zbyt późno.
Rysunek 6.2 przedstawia działanie pętli **do…while.** Program pokazany na listingu 6.6 ilustruje zastosowanie pętli **do...while**.
Program przedstawiony na tym listingu służy do wypisania na ekranie monitora żądanej ilości znaków oraz podaje wartości ich kodów. W liniach:

```
#define ST 96          //poczatek kodu
#define KONIEC 10      //ile znakow
```

ustalono numer kodu pierwszego znaku (kod 96 to litera a) oraz ilość wyświetlanych znaków (tutaj 10). W liniach:

```
do
    printf("\n%4i %4c  %5i",i++,ST+i,ST+i);
while (i <= KONIEC);
```

realizowana jest zasadnicza pętla **do…whlie**, w funkcji **printf()** następuje inkrementacji zmiennej **i** oraz wyświetlane są znaki i kody liter.




Rys.6.2 Budowa pętli **do…while**

Listing 6.6 Instrukcja do…while

```
#include <stdio.h>
#define ST 96           //poczatek kodu
#define KONIEC 10       //ile znakow
#define LINIA 22        //dlugosc linii
#define HORIZ 0xC4      //znak tworzacy linie
void main(void)
{ int k = 1, i =1;
  printf("\n   nr  znak  ma kod\n");
  do
   { putchar(HORIZ);
     k++;
    }
  while (k < LINIA);
  do
    printf("\n%4i %4c  %5i",i++,ST+i,ST+i);
  while (i <= KONIEC);
     k = 1;
     printf("\n");
  do
   { putchar(HORIZ);
     k++;
    }
  while (k < LINIA);
}
```

W liniach:

```
do
  { putchar(HORIZ);
    k++;
  } while (k < LINIA);
```

realizowane jest umieszczanie znaku semigraficznego 0xC4 na ekranie monitora – jest to pozioma kreska, zestaw 22 znaków kreśli linię.

## 6.4. Instrukcja for

Pętla **for** jest kolejną konstrukcją umożliwiającą organizację cyklu obliczeniowego. Stosujemy ją najczęściej wtedy, gdy znamy dokładnie liczbę powtórzeń, (ale nie jest to warunek konieczny). Jest to bardzo elastyczna konstrukcja, od dużych możliwościach. Spowodowane jest to faktem, że wyrażenia sterujące działaniem pętli **for** znajdują się w jednym miejscu.

Konstrukcja **for** ma następującą postać:

```
for (wyrażenie1; wyrażenie2; wyrażenie3)
   instrukcja
następna_instrukcja
```

oraz

```
for (wyrażenie1; wyrażenie2; wyrażenie3)
{
   instrukcja1
   instrukcja2
    ..............
   instrukcjN
}
następna_instrukcja
```

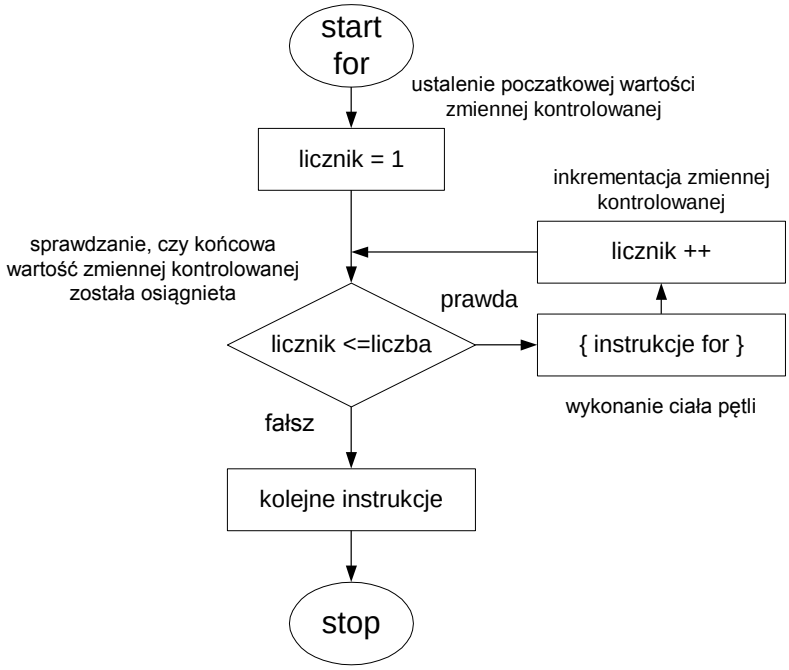
Równoważna postać pętli **for** wyrażona przy użyciu pętli **while** ma postać:

```
wyrażenie1;
while (wyrażenie2)
 { instrukcja
   wyrażenie3;
 }
następna_instrukcja
```

W nawiasie występujący po słowie kluczowym **for** zawarte jest wyrażenie pętli. Wyrażenie pętli składa się z trzech oddzielnych wyrażeń:

- wyrażenie ***inicjujące***
- wyrażenie ***testujące***
- wyrażenie ***inkrementujące***

Te trzy wyrażenia rozdzielone są średnikami. Konstrukcja pętli **for** pokazana jest na rys.6.3. Działanie pętli for jest następujące. Najpierw obliczane jest wyrażenie inicjujące (**wyrażenie1**). Służy ono do zainicjowania pętli. Następnie obliczane jest wyrażenie testujące (**wyrażenie2**). Jeżeli jest niezerowe (ma wartość prawda), wtedy wykonywana jest instrukcja, wyrażenie inkrementujące (**wyrażenie3**) jest obliczane i sterowanie jest ponownie przekazywane do pętli **for** (nie jest wykonywane już wyrażenie inicjujące). Ten proces wykonywany jest dopóki wyrażenie testujące (**wyrażenie2**) jest równe zeru (fałsz). Wtedy sterowanie przekazywane jest do następnej instrukcji (**nastepna_instrukcja**).




Rys.6.3. Sieć działań struktury powtarzania **for.**

Poniżej pokazujemy klasyczne wykorzystanie pętli **for.**

```
for (i = 1; i <= n;  ++i)
   czynnik *= i ;

for (j = 2; k % j == 0; ++j)
{
  printf(" \n%d jest podzielnikiem %d", j, k);
   suma += j;
}
```

Wyrażenie pętli może przybierać najróżniejsze formy - np. można opuścić wyrażenie inicjujące pętlę lub wyrażenie inkrementujące, ale zawsze muszą pozostać w nawiasach dwa średniki.

Rozważmy przykłady.

```
i = 1;
suma = 0;
for ( ; <= 10; ++i)
    suma += i;
```

Jeżeli opuścimy wyrażenie testujące, kompilator zakłada, że test ma wartość prawda i wtedy otrzymujemy pętlę nieskończoną.

Wyrażenie pętli może także zawierać inne wyrażenia:

```
for ( i = 0, suma = 0; i < 10; i++)
{
   suma +=i;
   printf("i = %d, suma = %d\n", i, suma);
}
```

Pętla **for** może być zagnieżdżona. Konstrukcja pętli zagnieżdżonej ma postać:

```
for (...........)
  for(.........)
    for(..............)
     instrukcja
```

Wykorzystamy pętlę **for** w programie (pokazanym na listingu 6.29),

Listing 6.6 .Obliczanie wartości wielomianu, pętla for

```
/*program oblicza wartosc wielomianu */
#include <stdio.h>
#include <conio.h>
int main()
{ float y,x,a,b,c;
  float z1,z2,dx;       //przedzial, przyrost
  printf("\npodaj wspolczynniki a , b, c  ");
  scanf("%f%f%f",&a,&b,&c);
  printf("\npodaj zakres z1, z2  ");
  scanf("%f%f",&z1,&z2);
  printf("\npodaj dokladnosc X : dx =  ");
  scanf("%f",&dx);
  printf("\nfunkcja y=%.2fx^2+ %.2fx + %.2f\n",a,b,c);
  printf("         x           y\n");
  for (x = z1; x < z2; x = x + dx)
   { y = a*x*x + b*x + c;
     printf("  %10.2f  %10.2f \n",x,y);
   }
  getch();
  return 0;
}
```

Zadaniem programu jest sporządzenie tabeli wartości wielomianu:

$$ax^2 + bx + c$$

Dla określonego przedziału wartości **x**:

$$z1 < x < z2$$

Po uruchomieniu programu należy wprowadzić z klawiatury współczynniki wielomianu oraz zakres zmienności x (zmienna **z1** i **z2**). Wartość wielomianu obliczana jest w pętli **for**. Pętla **for** ma postać:

```
for (x = z1; x < z2; x = x + dx)
   {
     y = a*x*x + b*x + c;
     printf("  %10.2f  %10.2f \n",x,y);
   }
```

Wyrażenie inkrementujące ma postać:

```
x = x + dx
```

Za każdym wykonaniem pętli wartość **x** zwiększa się o **dx.** Zmienna **x** przybiera wartości od **z1** do **z2.**

Przykładem użycia zagnieżdżonej pętli **for** jest kolejny (listing 6.7) program, tworzący tabliczkę mnożenia.

Listing 6.7 Tabliczka mnożenia, zagnieżdżona pętla for

```
/*program drukuje tabliczke mnozenia */
#include <stdio.h>
#include <conio.h>
int main()
{  int wx , ky;
   char l1 = '\xB3', l2 = '\xC4';
   window(31,5,51,6);
   textcolor(RED);
   textbackground(GREEN);
   cputs("Tabliczka mnozenia");
   printf("\n\n");
             printf("%7c",l1);
  for (ky=1;ky<=10;ky++)  printf("%6d",ky);
  putchar('\n');
  for (ky=1;ky<=70;ky++)  printf("%c",l2);
  putchar('\n');
  putchar('\n');
  for (wx = 1; wx <= 10; wx++)
    {printf("%6d%c",wx,l1);
     for (ky = 1; ky <= 10; ky++) printf("%6d",wx*ky);
      putchar('\n');
    }
  getch();
  return 0;
}
```

Fragment z zagnieżdżoną pętlą ma postać:

```
for (wx = 1; wx <= 10; wx++)
    {
     printf("%6d%c",wx,l1);
     for (ky = 1; ky <= 10; ky++) printf("%6d",wx*ky);
      putchar('\n');
    }
```

Program wykonuje 10 pętli zewnętrznych (dla zmiennej **wx**) oraz 10 pętli wewnętrznych ( dla zmiennej **ky**). W programie wykorzystano ***znaki semigraficzne*** do rysowania linii na ekranie:

```
char l1 = '\xB3', l2 = '\xC4';
printf("%7c",l1);
```

Znaki semigraficzne są wypisywane przez funkcję **printf()**, stosując kod szesnastkowy. Te kody zapisane w formacie:

```
\xB3   oraz   \xC4
```

są specyficzne dla środowiska PC, programy używające tej konwencji nie mogą pracować poprawnie w środowisku UNIX.
W programie wykorzystano także funkcje tekstowe środowiska Borland C++.

```
window(31,5,51,6);
textcolor(RED);
textbackground(GREEN);
cputs("Tabliczka mnozenia");
```

Zostanie utworzone okno tekstowe na ekranie oraz pojawi się w nim napis:

```
Tabliczka mnozenia
```

Funkcja

```
window(31,5,51,6);
```

definiuje rozmiary i położenie okna (zakładamy, że ekran pracuje w trybie 80x25 znaków). Funkcje:

```
textcolor(RED);
  textbackground(GREEN);
```

ustalają kolor napisu (**RED** - czerwony) oraz tła (**GREEN** - zielony). Gdy chcemy, aby napis migotał możemy dodać stałą do wartości koloru, np. zapis:

```
RED+BLINK
```

spowoduje migotanie czerwonego napisu. Do umieszczenia napisu w oknie tekstowym użyto funkcji **cputs().**
Pętla **for** jest elastyczna i ma bardzo duże możliwości w porównaniu z jej realizacjami w języku FORTRAN czy PASCAL. Następujące przykłady pokazują możliwości pętli **for**.

1. Klasyczne przykład pętli **for**:

```
for ( int i = 0; i < 100; i++)
```

licznik zainicjalizowany jest liczbą 0, granica licznika ustalona jest na mniej niż 100, zmienna kontrolowana **i** inkrementowana jest liczbą 1.

2. Użycie operatora dekrementacji, aby liczyć w dół:

```
for ( int i = 100; i >= 1; i--)
```

3. Modyfikowanie zmiennej kontrolowanej od 5 do 55 z krokiem 5:

```
for ( int i = 5; i <+= 55; i += 5)
```

4. Można stosować zmienne typu **char**:

```
char zn;
for ( zn = 'a'; zn <= 'z'; zn++)
```

5. Wyrażenie testujące może mieć złożoną postać:

```
for ( int i = 0; i*i*i <= 729; i++)
```

6. Wyrażenie inkrementujące może być dowolnym legalnym wyrażeniem:

```
int i;
int j = 55;
for ( int i = 1; j < 75; j =(++i * 5) + 50)
```

7. Wyrażenie sterujące może być puste:

```
int i;
int j = 2;
for ( i = 3; j <= 25;)  j = j *i;
```

Wartość zmiennej I cały czas wynosi 3, zmienna j startuje od wartości 2 i jest zmieniana w wyniku działania pętli. Pętla ta jest poprawna i skończona.

8. Pętla nieskończona, puste środkowe wyrażenie zawsze jest uznawane za prawdziwe:

```
for ( ; ; )  printf("pracuje bez konca \n");
```

Konstrukcja **for(; ;)** niekoniecznie musi tworzyć pętlę nieskończoną. Instrukcja **break** w dowolnym miejscu pętli powoduje natychmiastowe zakończenie:

```
for(; ;)
{
   zn = getchar();
   if ( zn == 'Q') break;
}
```

9. Wyrażenie inicjujące nie musi inicjalizować zmiennej:

```
int x;
for (printf(”Podaj liczbe \n”) ; x != 0; )
scanf(”%d”,&x);
```

W pętli **for** następuje wczytywanie liczb i drukowane są one na ekranie monitora dopóki nie wprowadzimy liczby 0.

10. Pętlą może sterować wiele zmiennych:

```
for (x=0, y=0; x + y < 10; ++x)
scanf(”%d”, &y);
```

Korzystając z tej własności można napisać efektywna pętle do odwrócenia napisu:

```
int i, j;
for (i=strlen(s)-1,j = 0; i >= 0;j++,i--) r[i] = s[j];
```

Napis „program” przyjmie postać „ margorp”.

11. Wyrażenie warunkowe może być dowolna operacja logiczną:

```
for ( i=0; i <3 && strcmp(str, ”wacek”) ; ++i)
{
   printf(“podaj haslo : “);
   gets(str);
}
```

W pokazanym przykładzie użytkownik podaje poprawne hasło (w tym przypadku jest to “**wacek**”). Pętla kończy się, gdy podane jest prawidłowe hasło lub podano trzy razy nieprawidłowe hasło. Funkcja biblioteczna **strcmp()** porównuje dwa napisy i zwraca wartość 0 w przypadku, gdy są jednakowe.

12. Pętle for często są stosowane, jako pętle opóźniające:

```
for (t = 0; t < X; t++);
```

W tej pustej pętli **for** realizowane jest opóźnienie, spowolnienie działania programu, gdzie X przyjmuje odpowiednio dużą wartość, gdy opóźnienie ma być znaczne.

# ROZDZIAŁ 7

## ZNAKI, ŁAŃCUCHY

## 7.1. Wstęp

Znaki są podstawowymi elementami, z których tworzy się programy źródłowe. Sekwencje znaków są odpowiednio grupowane i interpretowane tak, aby sterować pracą komputera. Łańcuch (lub napis) to grupa znaków traktowanych, jako całość. Język C nie obsługuje typu łańcuchowego. W języku C łańcuch jest tablicą znaków kończącą się znakiem zerowym (\0). Kolejne znaki tworzące łańcuch, znajdują się w kolejnych komórkach pamięci. Obsługa łańcuchów w języku C nie jest prostym zadaniem. Ponieważ obsługa znaków i łańcuchów jest ważna w aplikacjach, biblioteki standardowe języka C dostarczają wiele funkcji do obsługi znaków i łańcuchów.

## 7.2. Znaki

Program może zawierać stałe znakowe. Stała znakowa jest wartością całkowitą, przedstawianą, jako jeden znak umieszczony w apostrofach. Każdemu znakowi przyporządkowany jest kod ASCII. Typ **char** służy do przechowywania znaków takich jak litery (angielskie) czy znaki przestankowe. Typ **char** w rzeczywistości przechowuje liczby całkowite. W kodzie ASCII wartość 122 oznacza małą literę ‘z’. Standardowy kod ASCII potrzebuje 7 bitów do zapisu znaku, zakres wartości jest od 0 do 127. Typ **char** jest najczęściej definiowany, jako 8-bitowa jednostka pamięci. Kodowanie na 8 bitach pozwala na zapisanie 255 znaków. Dlatego wiele komputerów rozszerza standard ASCII, dostarczając większy zestaw znaków. Aby uwzględnić narodowe znaki **Inicjatywa Unicode** zaproponowała kod, który pozwala na reprezentowanie dużej ilości znaków, obecnie zawiera on ponad 40 000 znaków. Aby zapisać (zakodować) taką ilość znaków potrzeba 16 bitów. Sprzęt korzystający z kodu Unicode powinien posiadać typ **char** o długości 16 bitów. Należy jednak pamiętać, że standard języka C definiuje bajt jako liczbę bitów używanych przez typ **char**, w takim przypadku bajt miałby 16 bitów (a nie jak obecnie 8 bitów). Deklarowanie zmiennych typu **char** odbywa się tak samo jak w przypadku innych typów. Przykłady takich deklaracji to:

```
char zn;
char znak, litera;
```

Przypisanie znaku zmiennej odbywa się za pomocą instrukcji:

```
char zn = 'z';
```

Pojedyncza litera umieszczona miedzy znakami apostrofu jest stałą znakową. Gdy kompilator napotka zestaw znaków **‘z’**, zmieni go na odpowiednią wartość kodu ASCII. Deklaracja i przypisanie może mieć oczywiście standardowa postać:

```
char zn ;
zn = 'H' ;
```

Znaki przechowywane są, jako wartości liczbowe, zatem możemy inicjalizować zmienną znakową w następujący sposób:

```
char zn = 66 ;
```

co jest równoważne zapisowi :

```
char zn = 'B" ;
```

ponieważ kodem znaku 'B' jest liczba 66.
W programach w języku C do obsługi znaków i łańcuchów musimy używać plików nagłówkowych <ctype.h> oraz <string.h>. Należy pamiętać, że w języku C nie mamy kontroli czy nie używamy elementu spoza zdeklarowanego rozmiaru tablicy lub czy wartość nie została przekroczona dla konkretnego typu. Tego typy kontrola jest zadaniem programisty. Do funkcji znakowych parametry są przekazywane w postaci wartości typu **int,** wykorzystany jest tylko młodszy bajt. W takim przypadku występuje niejawna konwersja argumentów do typu **unsigned char**. Nic jednak nie stoi na przeszkodzie, aby argumenty przekazywać w postaci znakowej, ponieważ w takim przypadku wystąpi automatyczna konwersja typu znakowego do typu **int.** Oryginalne kody ASCII, które definiują znaki, zawierają się w zakresie od 0 do 127 (dziesiętnie). Znacznie później IBM wprowadził dodatkowo 128 znaków o kodach pomiędzy 128 a 255. Te nowe znaki to litery inne niż te, które występują w angielskim alfabecie oraz znaki semigraficzne. W języku C, liczba reprezentująca znak może być podana w postaci ósemkowej albo szesnastkowej. W systemach UNIX tradycyjnie stosujemy zapis ósemkowy, na komputerach osobistych najczęściej stosujemy notację szesnastkową. Dla ilustracji zastosowania znaków semigraficznych pokażemy program rysujący na ekranie monitora szachownicę.

Listing 7.1.drukowanie znaków semigraficznych

```
//szachownica
#include <stdio.h>
#include <conio.h>
int main()
{int x, y;
 printf("\n");
  for (y=1; y<9 ; y++)
   {
    for (x=1; x<9; x++)
      if ( (x+y) % 2 == 0)
         printf ("\xDB\xDB");
      else
         printf("  ");
    printf("\n");
   }
 getche();
 return 0;
}
```

W programie pokazanym na listingu 7.1 został użyty znak semigraficzny \xDB, który rysuje prostokąt. Zewnętrzna pętla **for**, która obsługuje zmienną **y**, po wykonaniu instrukcji przemieszcza kursor o jeden wiersz w dół. Wewnętrzna pętla **fo**r, która obsługuje zmienną **x** przemieszcza kursor w poprzek ekranu, zawsze o jedna kolumnę. Powstaje szachownica o wymiarach 8x8. Ponieważ na ekranie komputera osobistego znaki są zwykle dwa razy wyższe niż szersze, każdy kwadrat szachownicy zbudowany jest z dwóch wypełnionych prostokątów (znak \xDB).
W tabeli 7.1 pokazano funkcje biblioteczne, służące do obsługi znaków. Wymagany jest plik nagłówkowy <ctype.h>

Tabela 7.1 Funkcje obsługi znaków

| Nr | Nazwa | Działanie |
|---|---|---|
| 1 | isalnum | Zwraca wartość ≠ 0, gdy argument jest literą lub cyfrą |
| 2 | isalpha | Zwraca wartość ≠ 0, gdy argument jest literą |
| 3 | isblank | Zwraca wartość ≠ 0, gdy argument jest znakiem białym |
| 4 | iscntrl | Zwraca wartość ≠ 0, gdy argument znakiem z zakresu (0, 0x1F ) |
| 5 | isdigit | Zwraca wartość ≠ 0, gdy argument jest cyfrą |
| 6 | isgraph | Zwraca wartość ≠ 0, gdy argument jest znakiem drukowalnym innym niż spacja |
| 7 | islower | Zwraca wartość ≠ 0, gdy argument jest małą literą |
| 8 | isprint | Zwraca wartość ≠ 0, gdy argument jest znakiem drukowalnym, włączając w to spację |
| 9 | ispunct | Zwraca wartość ≠ 0, gdy argument jest znakiem interpunkcyjnym |
| 10 | isspace | Zwraca wartość ≠ 0, gdy argument jest białym znakiem |
| 11 | isupper | Zwraca wartość ≠ 0, gdy argument jest dużą literą |
| 12 | isxdigit | Zwraca wartość ≠ 0, gdy argument cyfrą szesnastkową |
| 13 | tolower | Zamienia dużą literę na małą |
| 14 | toupper | Zamienia małą literę na dużą |

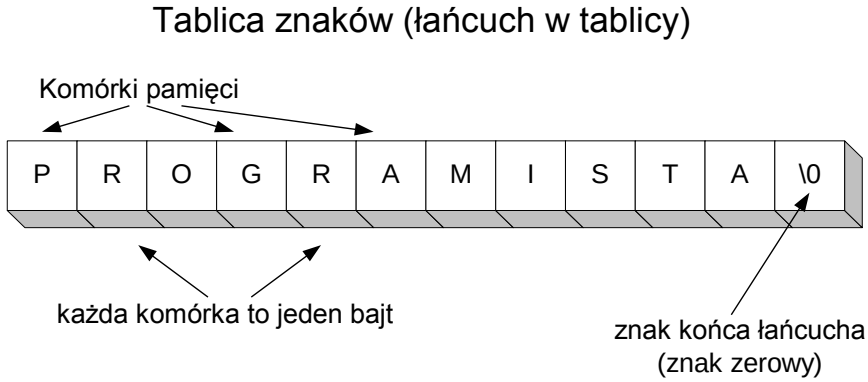
Pokazane w Tabeli 7.1 funkcje, mają wiele zastosowań. Na listingu 7.2 mamy program za pomocą, którego możemy wyświetlić na ekranie monitora zbiór znaków ASCII. Mamy umieszczony semigraficzny znak prostokąta przy znaku, który jest literą lub cyfrą. W programie wykorzystano dwie funkcje: **isprint()** oraz **isalnum().** Te dwie funkcje działają poprawnie tylko ze znakami ASCII o wartościach z zakresu od 0 do 127. Makro **isalnum** zwraca wartość niezerową, gdy argument został rozpoznany, jako litera albo cyfra. W przeciwnym przypadku makro zwraca wartość zero. Makro **isprint** sprawdza, czy znak jest drukowalny, włączając w to spację. Makro zwraca wartość różną od zera, gdy znak jest drukowalny, przeciwnym przypadku zwraca wartość zero. Znaki drukowalne należą najczęściej do przedziału od 0x20 do 0x7E.

Listing 7.2.Zasosowanie funkcji isprint() i isalnum()

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>
int main()
{ int zn, i, m=0xdb;
  for (i = 0, zn =0; zn <= 0x7f; zn++)
  { printf("%#04x ", zn);
    if(isprint(zn)) printf(" %c", zn);
    else
                    printf("  ");
    if (isalnum(zn) != 0)  printf(" %c ", m);
    else
                           printf("   ", zn);
    i++ ;
    if (i == 0)
    {printf(" \n ");
     i = 0;
    }
  }
  getche();
  return 0;
}
```

## 7.3. Łańcuchy

Język C nie posiada specjalnego typu łańcuchowego. Zamiast tego, łańcuchy są przechowywane w tablicach zbudowanych z elementów typu char. Biblioteka języka C do obsługi napisów udostępnia wiele użytecznych funkcji, które pozwalają efektywnie manipulować napisami. Znaki łańcucha są przechowywane razem, w kolejno po sobie następujących komórkach w pamięci. Model przechowywania łańcucha w pamięci pokazano na rysunku 7.1.




Rys.7.1. Dane typu łańcuchowego w pamięci, łańcuch ”PROGRAMISTA”

Zmienna o wartości ”PROGRAMISTA” zapisana jest w pamięci w postaci pokazanej na rys.7.1. Każdy znak przechowywany jest w oddzielnej komórce pamięci. Symbol **\0** jest specjalnym kodem, jaki język C wstawia na końcu łańcucha. Symbol **\0** jest nazywany **ogranicznikiem łańcucha** i traktowany jest jak jeden znak. Ogranicznik łańcucha, nazywane także **znakiem zerowym** nie jest cyfrą zero, jest on znakiem niedrukowalnym w kodzie ASCII równym 0. Łańcuchy w języku C są zawsze przechowywane razem z tym znakiem, co oznacza, że tablica znaków musi mieć długość przynajmniej o jedna komórkę większą niż długość zapisanego w niej łańcucha.
Stałą łańcuchowa możemy zadeklarować korzystając z dyrektywy preprocesora **#define** i cudzysłowów:

```
#define PROGRAMISTA ”Donald E.Knuth”
```

Znaki cudzysłowu nie są częścią łańcucha. Do deklaracji zmiennej łańcuchowej używamy zmiennej tablicowej typu **char** i podajemy liczbę znaków, które ta zmienna może zawierać. Składnia jest następująca:

```
char napis[N];
```

**napis** jest nazwą zmiennej (nazwą tablicy), **N** jest maksymalną liczbą znaków.
Stała łańcuchowa **”x”** nie jest tym samym, co stała znakowa **‘x’**. Stała znakowa **‘x’** należy do typu podstawowego **char**, natomiast **”x”** należy do typu pochodnego, tablicy typu **char**. Stała znakowa ‘x’ w pamięci zapisana jest w jednym bajcie, stała **”x”** to w zasadzie dwa znaki (potrzebne są dwa bajty do przechowania tej stałej), jeden to znak **‘x’** a drugi to znak zerowy **‘\0’.**
W programie 7.3 demonstrujemy zasady posługiwania się łańcuchami.

Listing 7.3.Łańcuchy

```
/* uzycie lancuchow */
#include <stdio.h>
#include <conio.h>
#define DL 30
#define DIALOG “Ach, jakie piekne imie”
int main()
{ char imie[DL];
  printf("Podaj swoje imie: \n");
  scanf(„%s”, imie);
  printf("Witaj, %s. %s\n", imie, DIALOG);
  getche();
  return 0;
}
```

Przebieg działania programu wygląda następująco:

```
Podaj swoje imie:
Wacek
Witaj, Wacek. Ach, jakie piekne imie
```

Do obsługi łańcucha funkcja **printf()** potrzebuje specyfikatora **%s.** W tablicy **imie[ ]** znak zerowy (**\0**) jest umieszczany automatycznie, zajmuje się tym kompilator. Należy zwrócić uwagę, w jaki sposób następuje odczytanie łańcucha przez funkcję **scanf().** Ponieważ zmienna jest typu tablicowego, nie używamy symbolu **&** przed nazwą zmiennej a także nie używamy nawiasów kwadratowych:

```
scanf(„%s”, imie);
```

Tablice znakowe możemy inicjalizować na kilka sposobów. Najbardziej popularne jest napisanie wprost łańcucha:

```
char napis[ ] = { ”Anna” } ;
```

W tej instrukcji wstawiamy do tablicy o nazwie **napis** znaki : ‘A’, ‘n’, ‘n’, ‘a’ oraz ‘\0’. Uproszczona wersja pokazanej inicjalizacji tablicy znakowej pozwala opuścić nawiasy klamrowe:

```
char napis[ ] =  ”Anna”  ;
```

Możemy inicjalizować tablice znakowe podając oddzielne znaki tworzące napis:

```
char napis[ ] = { ‘A’, ‘n’, ‘n’, ‘a’, ‘\0’ } ;
```

Rozmiar tablicy może być podawany jawnie, należy jednak pamiętać, aby rezerwowana liczba elementów była, co najmniej o jeden większa (łańcuch musi kończyć się znakiem ‘\0’). Jeżeli ilość znaków napisu jest mniejsza niż zarezerwowany rozmiar tablicy, niewykorzystane elementy tablicy otrzymają wartość ‘\0’. Należy pamiętać o fakcie, że rozmiar tablicy jest stały. Poprawna deklaracja tablicy ma postać:

```
#define ROZMIAR 10
char napis [ ROZMIAR ];
```

Nie możemy jawnie podawać rozmiar tablicy w następujący sposób:

```
int ROZMIAR = 10;
char napis [ ROZMIAR ];   //błąd
```

Możemy pisać stosunkowo proste programy pozwalające na manipulowanie łańcuchami. Należy jednak pamiętać, że biblioteka standardowa dostarcza dużą liczbę funkcji do operowania na napisach. Pokazane przez nas programy służą jedynie do ilustracji problemów. Pokazany na listingu 7.4 program służy do łączenia dwóch napisów. W łańcuchu wynikowym **n3** umieszczamy kolejno najpierw łańcuch zapisany w zmiennej **n1** a potem w zmiennej **n2.** Przepisywanie realizowane jest w pętli **for**. Ponieważ nie podajemy długości łańcucha, pętla do zakończenia działania wykorzystuje znacznik końca (‘\0’). W łańcuchu wynikowym **n3** musimy wstawić znak końca (‘\0’). W funkcji **main()** zadeklarowano tablicę **n3.** W naszym przykładzie można w niej umieścić 80 znaków. Do programisty należy ustalenie właściwego rozmiaru tej tablicy.

Listing 7.4.Łańcuchy, łączenie dwóch napisów

```
#include <stdio.h>
#include <conio.h>
int main()
{  const char n1[] = "suma ";
   const char n2[] = "napisow";
   char n3[81];
   int i,j;
   for (i=0; n1[i] != '\0'; ++i)
     n3[i] = n1[i];
   for (j=0; n2[j] != '\0'; ++j)
     n3[i + j] = n2[j];
   n3[i+j] = '\0';
   printf("%s\n", n3);
   getche();
   return 0;
}
```

Najczęściej używane funkcje służące do manipulowania napisami (definicje umieszczone są w pliku <string.h>) pokazane są w tabeli 7.2. Biblioteka **string.h** definiuje typ **size_t** oraz makro NULL (wskaźnik zerowy).

Tabela 7.2 Funkcje obsługi łańcuchów (często używane)

| Nr | Nazwa | Działanie |
|---|---|---|
| 1 | strcpy(s1,s2) | Kopiuje s2 do s1, zwraca s1 |
| 2 | strcat(s1,s2) | Dołącza s2 do końca s1, zwraca s1 |
| 3 | strncat(s1,s2,n) | Dołącza s2 do końca s1, aż osiągnie koniec łańcucha s2 lub skopiuje n znaków |
| 4 | strlen(s) | Zwraca liczbę znaków w s, nie wlicza znaku '\0' |
| 5 | strcmp(s1,s2) | Zwraca 0, jeżeli s1 i s2 są jednakowe, wartość < 0, jeżeli s1>s2, wartość > 0 jeżeli s1<s2 |
| 6 | strncmp(s1,s2,n) | Działa jak strcmp, ale porównuje co najwyżej n znaków |
| 7 | strchr(s1, ch) | Zwraca wskaźnik do pierwszego wystąpienia znaku **ch** w łańcuchu s1, gdy go nie znajdzie zwraca null |
| 8 | strstr(s1,s2) | Zwraca wskaźnik do pierwszego wystąpienia łańcucha s2 w łańcuchu s1, gdy go nie znajdzie zwraca null |

Na listingu 7.5 pokazujemy program, który wykorzystuje kilka funkcji z biblioteki standardowej. Do wyznaczenia długości łańcucha wykorzystano funkcję **strlen(n1)**, do porównania łańcuchów wykorzystano funkcję **strcmp(n1,n2),** do znalezienia łańcucha w łańcuchu wykorzystano funkcje **strstr(n2,"jan").**

Należy pamiętać, że funkcja **strcmp()** zwraca fałsz w przypadku, gdy badane łańcuchy są jednakowe, należy do wypisania odpowiedniego komunikatu użyć zaprzeczenia (operator !).

Listing 7.5.Łańcuchy, funkcje biblioteczna obsługujące napisy

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
int main()
{ const char n1[80] = "witaj ";
  const char n2[80] = "Lucjan";
  printf("\nrozmiar n1 to %d", strlen(n1));
  if (!strcmp(n1,n2))printf("\nnapisy sa identyczne");
   else
                printf("\nnapisy nie sa identyczne");
  if (strstr(n2,"jan"))  printf ("\nznaleziono jan");
  getche();
  return 0;
}
```

Po uruchomieniu program z listing 7.5 mamy komunikat:

```
rozmiar n1 to 6
napisy nie sa identyczne
znaleziono Jan
```

# ROZDZIAŁ 8

## FUNKCJE

## 8.1. Wstęp

**Funkcja** jest odrębnym fragmentem kodu programu, spełniającym określone zadania, jest blokiem instrukcji o specyficznej budowie. Wśród programistów popularne jest stwierdzenie, że funkcja jest "sercem i duszą programowania w języku C". Zazwyczaj program strukturalny napisany w języku C jest zbudowany z wielu funkcji.

## 8.2. Podstawowe informacje o funkcjach

W języku C program bardzo często składa się z wielu funkcji. Dzięki funkcjom skomplikowane zadanie obliczeniowe możemy podzielić na mniejsze zadania. Wszystkie funkcje mają ten sam priorytet wykonania. Nie wolno zagnieżdżać funkcji w funkcji. W programie musi się znajdować przynajmniej jedna funkcja (o nazwie zastrzeżonej) – **main().** Funkcja **main()** jest funkcją uprzywilejowaną - w tym sensie, że jest to funkcja, od której zaczyna się wykonywanie programu. Inne funkcje są wywoływane z funkcji **main()** lub z innych funkcji. Program może mieć funkcje umieszczone w jednym pliku lub w wielu. Dzięki funkcjom możemy lepiej zorganizować skomplikowany program - zadanie do rozwiązania podzielone może być na mniejsze podzadania. Zaleca się, aby funkcja nie zawierała zbyt dużo linii kodu. Zwyczajowo przyjmuje się, że elegancka funkcja nie zawiera więcej jak 60 linii. Każda funkcja ma taką samą strukturę jak funkcja **main().** Należy umieszczać prototypy funkcji. Po wywołaniu funkcji, wykonywane są instrukcje w niej umieszczone. Po zakończeniu zadania, sterowanie przekazywane jest do polecenia znajdującego się bezpośrednio po instrukcji wywołania funkcji. Przyjrzyjmy się następującej funkcji:

```
void komunikat(void)
{ printf(”nie mozna wykonac tego zadania”);
}
```

Bardzo istotny jest pierwszy wiersz definicji funkcji. Dostarcza on kompilatorowi następujące informacje:

- nazwa funkcji
- typ zwracanej przez funkcje wartości
- listę argumentów (parametrów) przekazywanych do funkcji

Aby zilustrować użycie funkcji pokażemy program demonstracyjny. Program nie rozwiązuje konkretnego zadania, jest zbiorem funkcji, które wykonają przewidywane zadania. Powstał program szkieletowy - konkretna postać funkcji będzie opracowana później - program ma za zadanie jedynie poprawnie wywołać funkcje.

Przewidujemy, że program wykona następujące zadania:

1) uruchomi funkcję do obsługi wprowadzanych danych
2) uruchomi funkcję przetwarzającą dane
3) uruchomi funkcję wyświetlającą wyniki
4) uruchomi funkcję pozwalającą czytać wyniki na ekranie (np. wykona tzw. "przytrzymanie ekranu")

Wymienione funkcje uruchamiane będą przez funkcję **main(),** poprawne wywołanie funkcji będzie sygnalizowane odpowiednim komunikatem.

Listing 8.1 Współdziałanie funkcji w programie

```
1  /* funkcje */
2  #include <stdio.h>
3  #include <conio.h>
4  void wprowadz_dane();
5  void przetwarzaj();
6  void podaj_wynik();
7  void przerwa();
8  int main()
9  {
10   clrscr();
11   wprowadz_dane();
12   przetwarzaj();
13   podaj_wynik();
14   przerwa();
15   return 0;
16 }     //koniec main
17 void wprowadz_dane()
18 {
19  printf("\n funkcja wprowadz_dane().");
20  printf("\nczytam dane z klawiatury");
21 }
22 void przetwarzaj()
23 {
24     printf("\n........funkcja przetwarzaj().....");
25     printf("\nprzetwarzam dane");
26 }
27 void podaj_wynik()
28 {
29   printf("\n........funkcja podaj_wynik().....");
30   printf("\npodaje wyniki");
31 }
32 void przerwa()
33 {
34   printf("\n........funkcja przerwa().........");
35   printf("\nNacisnij ENTER aby skonczyc");
36   getchar();
37 }
```

W wyniku działania tego programu mamy napis na ekranie monitora:

```
........funkcja wprowadz_dane()....
czytam dane z klawiatury
........funkcja przetwarzaj().....
przetwarzam dane
........funkcja podaj_wynik().....
podaje wyniki
........funkcja przerwa().........
Nacisnij ENTER aby skonczyc
```

Funkcja **main(),** od której zawsze zaczyna się wykonywanie programu ma postać:

```
8  void main()
9  {
10   clrscr();
11   wprowadz_dane();
12   przetwarzaj();
13   podaj_wynik();
14   przerwa();
15   return 0;
16 }     //koniec main
```

W linii 10 funkcja polecenie **clrscr()** "czyści" ekran. Następnie w liniach 11, 12, 13 i 14 umieszczono kolejne wywołania funkcji. Funkcje podejmować będą działania zgodnie z zaplanowanym porządkiem (wykonanie "z góry na dół"). Funkcję można wywołać z dowolnego miejsca programu, nawet z wnętrza innej funkcji.
Konstrukcje użytych funkcji nie różnią się od konstrukcji funkcji **main().** Pokazane tutaj funkcje są tzw. ***funkcjami prostymi*** - nie zwracają żadnej wartości. Wywołanie funkcji jest realizowane po prostu przez umieszczenie jej nazwy razem z nawiasami okrągłymi. Nawiasy są konieczne, aby poinformować kompilator, że mamy do czynienia z funkcją a nie np. ze zmienną. Takie wywołanie funkcji jest instrukcją, wobec tego kończy się średnikiem.
Wywołanie:

```
11   wprowadz_dane();
```

powoduje przekazanie sterowania do kodu w definicji funkcji (linie 17 - 21). Funkcja wykonuje zadanie, kończy działanie a sterowanie powraca do funkcji **main()** i wykonuje się następna instrukcja (u nas linia 12).
Istnieją trzy zagadnienia związane z użyciem funkcji w programie:

- prototyp funkcji (deklaracja)
- definicja funkcji
- wywołanie funkcji

## 8.3. Deklaracja funkcji

W pokazanych na listingu 8.1 liniach 4 - 7 znajdują się ***deklaracje funkcji*** (**prototypy**). W języku C obowiązuje zasada, że każda funkcja przed jej użyciem musi być zadeklarowana. Formalnie deklaracja funkcji ma postać:

```
typ nazwa(deklaracje argumentów);
```

W deklaracji funkcji mamy trzy elementy:

- typ zwracany
- nazwa funkcji
- wykaz argumentów

Przykłady prototypów funkcji:

```
void fun1(void) ;
void  fun2( ) ;
int fun3( );
int fun4(char c) ;
double fun5(int x= 20, int y=20) ;
```

Typ **void** jest bardzo przydatny przy deklaracji funkcji lub przy opisie wskaźników do dowolnego typu danych. Jeżeli funkcja nie zwraca niczego prototyp ma postać:

```
void funX( ) ;
```

Jeżeli funkcja jest bezparametrowa, listę parametrów formalnych zastępuje **void**:

```
int getchar(void) ;
```

Praktycznie zapisy:

```
void fun1(void) ;
```

oraz

```
void  fun2( ) ;
```

są równoważne.

## 8.4. Definicja funkcji

Struktura funkcji ma następującą postać:

```
typ nazwa (deklaracja argumentów)
{
   blok instrukcji
   instrukcja return
}
```

Podobnie jak w deklaracji funkcji, w ***definicji funkcji*** podawany jest typ zwracany przez funkcję. Następnie występuje nazwa funkcji i para nawiasów okrągłych. Nie stawiamy średnika po nawiasach.

Instrukcje, tworzące ciało funkcji, zamknięte są nawiasami klamrowymi. Ostatnią instrukcją przed nawiasem klamrowym powinna być instrukcja **return** (zależy to od typu kompilatora). Jedynie dla funkcji typu **void** instrukcję **return** można opuścić. Deklaracja argumentów oznacza umieszczenie niezbędnych argumentów formalnych. Argumentami formalnymi mogą być:

- zmienne wszystkich typów podstawowych
- struktury
- unie
- wskaźniki
- referencje
- zmienne typów definiowanych przez użytkownika

Argumentami formalnymi nie mogą być tablice (ale mogą być wskaźniki do nich).

## 8.5. Wywoływanie funkcji

Wywołanie funkcji polega na przekazaniu sterowania do funkcji – komputer wtedy wykonuje instrukcje w niej zawarte. Po wykonaniu funkcji, sterowanie przekazywane jest z powrotem do funkcji **main().** Najczęściej wywołanie funkcji jest poleceniem obliczenia wartości wyrażenia, zwracanej przez nazwę funkcji. Aby wywołać funkcję należy umieścić w programie odpowiednią instrukcję. Składnia wywołania funkcji ma następującą postać:

```
nazwa_funkcji (argumenty aktualne) ;
```

Liczba, kolejność i typy argumentów aktualnych powinny zgadzać się z argumentami formalnymi, występującymi w deklaracji i definicji funkcji.

## 8.6. Zmienne w funkcjach

Istotnym zagadnieniem podczas opracowywania programu jest rozważenie gdzie i jak będą zadeklarowane zmienne. W języku C istnieje wiele różnych typów zmiennych, każda zmienna ma dwa atrybuty: typ i klasę pamięci (ang. **storage class**).

Mamy następujące klasy:

- zmienne automatyczne (lokalne) - **auto**
- zmienne zewnętrzne (globalne) - **extern**
- zmienne statyczne - **static**
- zmienne rejestrowe - **register**

**Zmienne automatyczne (lokalne).**

Zmienne zadeklarowane wewnątrz ciała funkcji są ***zmiennymi automatycznymi*** (lokalnymi).

Rozważmy fragment kodu:

```
{
  int a, b,c;
  int k;
  a = 1; b = 2; k = 10;
  c = (a + b)*k;
}
```

Gdy sterowanie programem dochodzi do bloku, system rezerwuje odpowiednią ilość pamięci dla automatycznie zadeklarowanych zmiennych. Zmienne **a, b, c, k** są zadeklarowane wewnątrz bloku i są lokalnymi względem bloku. Gdy sterowanie opuszcza blok, system zwalnia pamięć zarezerwowaną na te zmienne - wartości zmiennych są tracone. Jeżeli sterowanie ponownie będzie przekazane do tego bloku, pamięć od nowa jest przydzielana, ale poprzednie wartości nie są znane. Jeżeli w definicji funkcji znajduje się blok, każde wywołanie funkcji powoduje nowe tworzenie środowiska dla zmiennych. Ponieważ zmienna lokalna działa w obrębie swojej funkcji, nie jest widoczna przez inne funkcje. Tym samym zmienne umieszczone w dwóch różnych funkcjach mogą mieć te same nazwy, (ale mają inne znaczenia). Lokalność zmiennych powoduje, że zmienne zadeklarowane w jednej funkcji praktycznie nie mogą być wprost dostępne przez inne funkcje.

**Zmienne globalne.**

Jednym ze sposobów przekazywania informacji pomiędzy blokami i funkcjami jest stosowanie ***zmiennych zewnętrznych*** (globalnych). Gdy zmienna jest zadeklarowana poza funkcją, przydzielona jest jej permanentna pamięć, ta pamięć należy do klasy **extern.** Typowo w programie należy zadeklarować zmienną na zewnątrz funkcji.

Listing 8.2.Funkcje, zmienne globalne

```
#include <stdio.h>
double x = 13.0;        //zmienna globalna
double y = 133.33;      //zmienna globalna
double war;             //zmienna globalna
int main()
{ double  fun();
  war = fun();
  printf("\n %f %f %f ",x,y,war);
  return 0;
}

double fun()
{
  double y,war:         // y i war są lokalne
  x = y = war = 13.0;
  return (x + y + war);
}
```

Jeżeli nasz program jest napisany w postaci dwóch plików, a zmienna o nazwie np. **dana_v** jest zadeklarowana, jako globalna w pliku pierwszym:

```
int dana_v;
```

a funkcja w pliku drugim potrzebuje tej zmiennej to należy w tej funkcji umieścić deklarację, korzystając ze słowa kluczowego **extern** :

```
extern int dana_v;
```

W poniższym programie zmienna **n** jest zmienną globalną. Funkcja **void wprowadz_dane()** czyta wartość **n** z klawiatury, ponieważ ta zmienna jest globalna, funkcja **main()** może wykorzystać wartość tej zmiennej.

Listing 8.3.Funkcje, zmienne globalne

```
#include <stdio.h>
#include <conio.h>
int n;
void wprowadz_dane();
void przerwa();
int main()
{ wprowadz_dane();
  printf("\nilosc danych  %d",n);
  przerwa();
  return 0;
}     //koniec main
void wprowadz_dane()
{
 printf("\nile danych wprowadzic  ");
 scanf("%d",&n);
}
void przerwa()
{
 printf("\nNacisnij ENTER aby skonczyc");
 getche();
}
```

Po uruchomieniu programu, program pyta, ile danych wprowadzić. Po wpisaniu liczby, np. 10, program podaje komunikat:

```
ilosc danych 10
Nacisnij ENTER aby skonczyc
```

**Zmienne klasy "register".**

Tego typu zmiennych używamy, gdy chcemy przyspieszyć szybkość wykonywania programu. Deklaracja zmiennej, jako **register** informuje kompilator, że jeżeli jest to możliwe fizycznie i semantycznie, to należy zadeklarowaną zmienną zapamiętać w rejestrach. Pojemność rejestrów jest niewielka i często nie jest możliwe wykonanie tego zlecenia, wtedy przydzielana jest pamięć konwencjonalna.

Zazwyczaj programista wybiera kilka zmiennych, często używanych w programie i deklaruje je, jako **register**. Typowe użycie takich zmiennych pokazane jest w następującym fragmencie kodu:

```
{  register int k;
   for (k = 0; k < 10; k++)
    { blok instrukcji
    }
} // po wyjściu z tego bloku, pamięć jest zwalniana
```

Należy pamiętać, że deklaracja **register** jest tylko sugestią dla kompilatora, aby zmienna była tego typu.

**Zmienne statyczne.**

***Zmienne automatyczne*** istnieją w funkcji tak długo jak długo wykonuje się funkcja, po zakończeniu działania funkcji, zmienne te po prostu są niszczone, a ich wartości są tracone. Technicznie rzecz biorąc zmienne używane przez funkcje są odkładane na stosie w momencie wywołania funkcji i zdejmowane ze stosu, gdy funkcja kończy działanie, innymi słowy, gdy sterowanie przekazywane jest do funkcji **main().** Można zapobiec utracie wartości zmiennej lokalnej, jeżeli zmienną taką zadeklarujemy, jako zmienną typu **static**. Zmienną typu **static** deklarujemy następująco:

```
static typ_zmiennej nazwa_zmiennej;
```

W wielu przypadkach chcemy, aby funkcja pamiętała jakieś informacje pomiędzy jej wywołaniami (chcemy, aby wartość lokalnej zmiennej w funkcji po jej ponownym wywołaniu pamiętała poprzednią wartość). Ten problem można rozwiązać za pomocą zmiennej globalnej, należy jednak pamiętać, że stosowanie zmiennych globalnych jest potencjalnym źródłem błędów. Wygodniejsze jest stosowanie zmiennych statycznych.
Zadaniem programu z listingu 8.4 jest poinformowanie użytkownika ile razy została wywołana funkcja.
Po uruchomieniu tego programu mamy następujący wynik na ekranie:

```
Podaj ile razy wywolac funkcje : 5
Petla for wywolala funkcje       5 razy
Teraz funkcja jest wywolana      1 raz
W sumie funkcja wywolana         7 razy
```

W podanym przykładzie, z klawiatury wprowadzamy liczbę 5. Pętla **for** wywoła funkcję **funS()** 5 razy. Następnie funkcja jest wołana jeden raz. Funkcja biblioteczna **printf()** wywołuje ponownie funkcję **funS(),** w sumie było 7 wywołań funkcji **funS().**W funkcji **funS()** zmienna **n** została zadeklarowana jako **static**. Wobec tego wartość tej zmiennej nie jest tracona. Kolejne wywołanie funkcji **funS()** zwiększa tą zmienną o 1. Na koniec funkcja **printf()** podaje ile razy funkcja **funS()** była wywołana.

Program z listing 8.4 pokazuje użycie zmiennej statycznej.

Listing 8.4.Funkcje, zmienne statyczne

```
#include <stdio.h>
#include <conio.h>
int funS(void)
  { static int n;
    return (++n);
  }
int main()
{ int i, ile;
  printf("\nPodaj ile razy wywolac funkcje : ");
  scanf("%d",&ile);
  for (i = 0; i < ile; i++)      funS();
  printf("\Petla for wywolala funkcje  %d razy",ile);
  printf("\nTeraz funkcja jest wolana   1 raz");
  funS();
  printf("\nW sumie funkcja wywolana %d razy",funS());
  getche();
return 0;
}
```

## 8.7. Funkcje zwracające wartość i stosowanie instrukcji return.

Jak już wspominaliśmy, wartość obliczana przez funkcje jest najczęściej przekazywana do wywołującego funkcję środowiska za pomocą instrukcji **return**.
Instrukcja **return** może być stosowana na dwa sposoby:

| | | |
|---|---|---|
| jako: | return ; | ( nie udostępnia wyniku ) |
| lub jako: | return wyrażenie; | ( przekazuje wartość wyrażenia) |

Przykłady użycia instrukcji **return** są następujące:

```
return;
return(0);
return (13);
return (x + y);
return (++n)
```

Instrukcja **return** spełnia dwa zadania:
1) wykonanie jej natychmiast przekazuje sterowanie z funkcji do środowiska wywołującego funkcję
2) wartość wyrażenia zamkniętego w nawiasy znajdujące się za zapisem **return** jest zwracana, jako wartość funkcji do środowiska wywołującego

Instrukcja **return** nie musi znajdować się na końcu funkcji, można umieścić kilka instrukcji **return** w funkcji. Jeżeli wyrażenie związane z **return** jest innego typu niż specyfikator funkcji - nastąpi konwersja typu zmiennej do typu specyfikatora funkcji. Zilustrujemy to przykładem.
Mamy następujący fragment funkcji:

```
double funP(x,y)
int x,y;
{   int n;
.................
  return (n);
}
```

Zmienna **n** jest typu **int**, funkcja ma zwrócić wartość typu **double**, wobec tego zmienna **n** będzie przekonwertowana do wartości typu **double**. Można stosować wielokrotnie instrukcję **return**. Poniższy przykład ilustruje taką możliwość:

```
float funP(x)
  float x;
{  if (x >= 0.0)    return (x);
   else             return (-x);
}
```

Program demonstrowany poniżej wykonuje takie samo zadanie jak program z listingu 8.3, z tym, że nie używa zmiennej globalnej.

Listing 8.5.Funkcje, zmienne globalne nie są stosowane

```
#include <stdio.h>
#include <conio.h>
int wprowadz_dane();
void przerwa();
int main()
{ int n;
  n = wprowadz_dane();
  printf("\nilosc danych  %d",n);
  przerwa();
  return 0;
}     //koniec main
int wprowadz_dane()
{ int n;
 printf("\nile danych wprowadzic  ");
 scanf("%d",&n);
 return (n);
}
void przerwa()
{ printf("\nNacisnij ENTER aby skonczyc");
 getche();
}
```

Po uruchomieniu tego programu, otrzymujemy postać wyniku na ekranie tak jak w przypadku uruchomienia programu z listingu 8.3. W programie mamy teraz nowy prototyp funkcji **wprowadz_dane ()**:

```
int wprowadz_dane();
```

Tak zapisany prototyp informuje kompilator, że wartość zwracana będzie typu **int**. W definicji funkcji nastąpiła też zmiana, zastosowano instrukcję **return** :

```
return (n);
```

Funkcja zwróci wartość **n** (w naszym przypadku ilość danych do wprowadzenia). Należy jednak pamiętać, że instrukcja **return** może zwrócić tylko jedną wartość. Jeżeli chcemy, aby funkcja zwracała więcej wartości do środowiska wywołującego, należy użyć całkiem innych mechanizmów.
Kolejnym przykładem może być program, który potrzebuje funkcji zwracającej wartość typu **float**. Zadaniem programu jest obliczenie powierzchni kuli na podstawie podanego promienia. Powierzchnię oblicza funkcja **pow().**

Listing 8.6.Funkcje, obliczanie powierzchni kuli

```
#include <stdio.h>
#include <conio.h>      // dla getche()
#include <process.h>    // dla exit()
float pow(void);         //prototyp funkcji
int main()
{float s_kuli;
 s_kuli = pow();   //wywolanie funkcji
 printf("\npowierzchnia kuli = %.1f",s_kuli);
 getche();
 return 0;
}
float pow(void)
{ float r;
  printf("\npodaj promien kuli : ");
  scanf("%f",&r);
  if ( r < 0.0)
    { printf("\nzla wartosc");
      exit(0);
    }
  return(4.0 * 3.14159 * r * r);
}
```

Ponieważ funkcja **pow()** ma zwracać wartość typu **float**, jej prototyp ma postać:

```
float pow(void);    //prototyp funkcji
```

Funkcja **main()** wywołuje funkcje pow():

```
s_kuli = pow();   //wywolanie funkcji
```

Funkcja **pow()** wymaga podania wartości promienia kuli a następnie w instrukcji **return** obliczana jest powierzchnia kuli:

```
return(4.0 * 3.14159 * r * r);
```

Obliczona wartość typu **float** zwracana jest do funkcji **main()** i drukowana jest wartość:

```
s_kuli = pow();   //wywolanie funkcji
pritf("\npowierzchnia kuli = %.1f",s_kuli);
```

## 8.8. Wywołanie funkcji, przekazywanie argumentów przez wartość

Funkcja jest wykonywana po wywołaniu jej przez środowisko (najczęściej jest to funkcja **main()**). Wywołanie funkcji w programie polega na napisaniu nazwy funkcji oraz umieszczeniu listy argumentów w nawiasach. Typowo, liczba argumentów i typy zmiennych odpowiadają argumentom wymienionym w definicji funkcji. Wszystkie argumenty są przekazywane przez wartość.

***Wywołanie funkcji*** oznacza wykonanie szeregu czynności:

- każde wyrażenie w liście argumentów jest wyliczone
- wartość wyrażenia jest przypisana odpowiadającemu parametrowi formalnemu na początku ciała funkcji
- ciało funkcji jest wykonywane
- jeżeli instrukcja **return** jest wykonana, sterowanie przekazywane jest natychmiast do środowiska wywołującego funkcję
- jeżeli w instrukcji **return** znajduje się wyrażenie, wtedy wartość wyrażenia jest konwertowana, (jeżeli jest to niezbędne) do typu, jaki jest wymieniony w specyfikatorze typu funkcji, a wartość jest przekazywana do środowiska wywołującego funkcję
- jeżeli w funkcji nie ma instrukcji **return,** sterowanie jest przekazywane do środowiska wywołującego funkcję po napotkaniu nawiasu klamrowego, zamykającego ciało funkcji
- jeżeli w funkcji znajduje się instrukcja **return** bez wyrażenia lub nie ma instrukcji **return**, wtedy żadna użyteczna wartość nie jest przekazana z funkcji do środowiska wywołującego
- wszystkie argumenty są przekazywane przez wartość

Przekazywanie argumentów "***przez wartość***" odróżniamy od przekazywania argumentów przez „***referencję”*** (ang. *call by reference*). Ten ostatni sposób oznacza przekazywanie adresu zmiennej (**referencji**) do funkcji.
Na początku opiszemy prosty program wykorzystujący funkcję obliczającą pierwiastek kwadratowy (należy pamiętać, że w bibliotece jest odpowiednia funkcja o nazwie **sqrt()**). Do funkcji o nazwie **pierw()** przekażemy argument (liczbę, z której chcemy obliczyć pierwiastek kwadratowy).

Ciekawostką jest metoda obliczania pierwiastka kwadratowego - pierwiastek kwadratowy z dodatniej liczby obliczamy z zależności rekurencyjnej:

$$x_{n+1} = 0.5\left( x_n + \frac{a}{x_n} \right)$$

$$x_n \to \sqrt{a}\,, \text{dla } n \to \infty$$

Jako pierwsze przybliżenie ($\mathbf{x_1}$) przyjmuje się 1. Proces obliczeniowy kończymy, gdy wartość obliczanego pierwiastka osiągnie żądaną dokładność.

Listing 8.7.Funkcje, przekazywanie argumentów

```
//funkcja oblicza pierwiastek kwadratowy
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <process.h>
#define PREC 0.01
double pierw(double);   //prototyp funkcji
int k= 1;

int main()
{double x,sqx;
 printf("\n podaj liczbe > 0  :");
 scanf("%lf",&x);
 sqx = pierw(x);
 printf("\n pierwiastek kwadratowy z %lf= %lf",x,sqx);
 printf("\nliczba iteracji %d",k);
 getche();
 return 0 ;
}
double pierw(double a)
{ double x = 1.0;
  while (fabs (x*x-a) > PREC)
    { x = (x + (a/x))*0.5;
      k = k+1;
    }
  return(x);
}
```

Po uruchomieniu programu otrzymujemy komunikat:

```
Pierwiastek kwadratowy z 25.000000 = 5.000023
Liczba iteracji 6
```

W programie funkcja **pierw()** jest zadeklarowana następująco:

```
double pierw(double);   //prototyp funkcji
```

Funkcja zwraca wartość typu **double**, przekazywany parametr jest także typu **double**. Funkcja jest wywoływana następująco:

```
sqx = pierw(x);
```

Użytkownik wpisuje z klawiatury wartość zmiennej **x**. W wywołaniu funkcji **pierw()** wartość ta jest przekazywana do niej, jako argument. Argumenty w definicji funkcji i w wywołaniu funkcji mają różne nazwy. Argument w wywołaniu funkcji (tu **x**) nosi nazwę ***argumentu aktualnego*** (inaczej faktycznego) a w definicji funkcji - ***argumentu formalnego*** (tu **a**).

## 8.9. Funkcje i tablice

Bardzo często stosujemy funkcje do obsługi tablic. Całą tablicę można przekazać do funkcji, ale należy pamiętać, że w języku C przekazywanie tablic jest nieco odmienne niż przekazywanie zmiennych innych typów. Jeżeli przekazujemy zwykłą zmienną, tworzona jest kopia danych i umieszczana w odpowiednim miejscu pamięci. Mamy dwa egzemplarze danych - ma to taką zaletę, że funkcja manipulując na przesłanej zmiennej, nie jest w stanie zmienić oryginalnej zmiennej. W przypadku przesyłania tablic do funkcji, przekazywany jest ***adres tablicy***. Nie jest tworzona kopia danych, nazwie drugiej tablicy przypisany jest ten sam obszar adresowy a więc i dane. Oczywiście najprostszym sposobem dostępu do danych tablicowych jest zadeklarowanie tablicy, jako zmiennej globalnej. Ilustruje to przykład z listingu 8.8. W tym przykładzie dane tablicowe są wczytywane przez funkcję - tworzona jest tablica o nazwie **dane.** Ponieważ tablica jest globalna, elementy tablicy są dostępne w funkcji **main().** Po uruchomieniu programu mamy następujący wydruk:

```
Ile danych wprowadzic   4
Podaj dane #0 : 11
Podaj dane #1 : 12
Podaj dane #2 : 13
Podaj dane #3 : 14

Ilosc danych  4
11
12
13
14

Nacisnij ENTER aby skonczyc
```

W programie zadeklarowano dwie zmienne globalne:

```
int n;
int dane[ROZ];    //tablica globalna
```

Funkcja **main()** wywołuje funkcję:

```
wprowadz_dane();
```

która zaczyna działanie, użytkownik wprowadza elementy tablicy.

Listing 8.8.Funkcje, obsługa tablic globalnych

```
#include <stdio.h>
#include <conio.h>
#define ROZ 100
int n;
int dane[ROZ];          //tablica globalna
void wprowadz_dane();
void przerwa();
int main()
{ int nr=0;
  wprowadz_dane();
  printf("\nilosc danych  %d \n",n);
  do                    //wydruk tablicy dane[]
    { printf("%d\t%d\n",nr,dane[nr]);
      nr++;
     } while(nr<n);
  przerwa();
  return 0;
}     //koniec main
void wprowadz_dane()
{ int i;
  printf("\nile danych wprowadzic  ");
  scanf("%d",&n);
  for (i=0; i<n; i++)
     { printf("Podaj dane #%d: ",i);
       scanf("%d",&dane[i]);
     }
}
void przerwa()
{printf("\nNacisnij ENTER aby skonczyc");
 getche();
}
```

Po skończeniu działania tej funkcji, sterowanie jest przekazywane do funkcji **main()** i wykonywane są kolejne instrukcje.

```
printf("\nilosc danych  %d \n",n);
  do              //wydruk tablicy dane[]
    { printf("%d\t%d\n",nr,dane[nr]);
      nr++;
    } while(nr<n);
```

W pętli wykonywane jest drukowanie elementów tablicy. Następny program (listing 8.9) ilustruje użycie kilku funkcji operujących na tablicach globalnych.

Wprowadzanie danych następuje z poziomu funkcji **wprowadz_dane()** - tworzona jest tablica globalna **dane**. Następnie z poziomu funkcji **main()** drukowana jest tablica. Z tej tablicy korzysta funkcja **obliczaj()** - zadaniem jej jest wyszukanie największego i najmniejszego elementu w tablicy oraz obliczenie średniej z wprowadzonych danych.

Listing 8.9.Funkcje, tablice globalne

```
#include <stdio.h>
#include <conio.h>
int n, dane[100];    //tablica globalna
void wprowadz_dane();
void obliczaj();
int main()
{ int nr=0;
  wprowadz_dane();
  printf("\nilosc danych  %d \n",n);
  do          //wydruk tablicy dane[]
    {  printf("%d\t%d\n",nr,dane[nr]);
       nr++;
    }   while(nr<n);
  obliczaj();
  getche();
  return 0;
}     //koniec main
void wprowadz_dane()
{ int i;
 printf("\nile danych wprowadzic  ");
 scanf("%d",&n);
 for (i=0; i<n; i++)      {
       printf("Podaj dane %d: ",i);
       scanf("%d",&dane[i]);
     }
}
void obliczaj()
{ float d_sr, suma = 0;
  int maks, mini, i = 0;
  maks = mini = dane[0];
  do
   {if (dane[i] > maks)  maks = dane[i];
    if (dane[i] < mini)  mini = dane[i];
    suma += dane[i];
    i++;
   }   while (i< n);
  d_sr = suma/n;
  printf("\n srednia  = %8.2f",d_sr);
  printf("\n maksimum = %6d",maks);
  printf("\n minimum  = %6d",mini);
 }
```

Po uruchomieniu programu mamy następujący wydruk:

```
Ile danych wprowadzic  5
Podaj dane 0 : 13
Podaj dane 1 : 15
Podaj dane 2 : 12
Podaj dane 3 : 10
Podaj dane 4 : 3
Ilosc danych  5
13
15
17
10
3
srednia = 11.60
maksimum = 17
minimum = 3
Nacisnij ENTER aby skonczyc
```

Tablice możemy przekazywać, jako argument (program 8.10).

Listing 8.10.Funkcje, przekazywanie tablicy

```
#include <stdio.h>
#include <conio.h>
#define ILE 5
void zmiana(int [ ]);
int main()
{int tem[ILE];
 int i;
 printf("\n wprowadz temperature w stopniach
         Fahrenheita \n");
 for (i=0; i< ILE; i++)
   {printf("podaj temperature %d :",i);
    scanf("%d",&tem[i]);
   }
 zmiana(tem);
 getche();
 return 0;
}  // koniec main()
void zmiana(int t[ ])
{int i;
 float celsius;
 printf("\nT [F]    T [C]\n");
 for (i=0; i<ILE; i++)
   {celsius = (5.0/9.0)*(t[i] - 32);
    printf("%4d\t%6.2f\n",t[i],celsius);
   }
}
```

Wymaga to odpowiedniej deklaracji funkcji i odpowiedniego wywołania - w wywołaniu przekazywana jest tylko nazwa tablicy. W programie (listing 8.10) zamieniamy temperaturę w stopniach Fahrenheita na temperaturę w stopniach Celsjusza. Konwersji temperatury dokonuje funkcja po przesłaniu do niej wprowadzonej tablicy. Po uruchomieniu programu mamy następujący wydruk:

```
Wprowadz temperature w stopniach Fahrenheita
podaj temperature 0 : 50
podaj temperature 1 : 60
podaj temperature 2 : 70
podaj temperature 3 : 80
podaj temperature 4 : 90
       T [F]   T [C]
        50     10.00
        60     15.56
        70     21.11
        80     26.67
        90     32.22
```

Kolejny program ilustruje przekazywanie tablicy typu **float** do funkcji.

```
Listing 8.11.Funkcje, tablice typu float
#include <stdio.h>
#include <conio.h>
#define ROZ 100
int n;       //zmienna globalna
void wprowadz_dane(float d[]);
void obliczaj(float d[]);
int main()
{float dane[ROZ];
 int nr = 0;
 wprowadz_dane(dane);
 printf("\nilosc danych  %d \n",n);
 do          //wydruk tablicy dane[]
    { printf("%d\t%10.2f\n",nr,dane[nr]);
      nr++;
    }  while(nr<n);
 obliczaj(dane);   getche();
 return 0;
}     //koniec main

void wprowadz_dane(float d[])
{int i;
 printf("\nile danych wprowadzic  ");
 scanf("%d",&n);
 for (i=0; i<n; i++)
     {
       printf("Podaj dane %d: ",i);
       scanf("%f",&d[i]);
     }
```

```
void obliczaj(float d[])
{ float d_sr,maks,mini;
  float suma = 0;
  int i=0;
  maks = d[0];    mini = d[0];
  do
   {if (d[i] > maks)    maks = d[i];
    if (d[i] < mini)    mini = d[i];
    suma += d[i];
    i++;
   }  while (i< n);
  d_sr = suma/n;
  printf("\n srednia  = %8.2f",d_sr);
  printf("\n maksimum = %8.2f",maks);
  printf("\n minimum  = %8.2f",mini);
 }
```

Program z listingu 8.11 działa podobnie jak program z listingu 8.9, jedyna różnica to zmiana typu wprowadzanych danych tablicowych z typu **int** na typ **float**.

## 8.10. Makrodefinicje

Makrodefinicja jest szczególną konstrukcją, charakterystyczną dla języka C. W zasadzie makrodefinicja może działać jak funkcja. Makrodefinicje tworzone są za pomocą dyrektywy preprocesora **define**. Jak wiemy, za pomocą tej dyrektywy tworzone są stałe, np.:

```
#define PI 3.14159
```

Jeżeli użyjemy tej dyrektywy w programie, kompilator, wszędzie tam gdzie znajdzie w programie nazwę stałej **PI** zamieni ją na wartość 3.14159. W języku C zakres stosowania dyrektywy **#define** jest szeroki - mamy możliwość użycia argumentów.
Konstrukcja makrodefinicji ma postać:

```
#define NAZWA (lista_parametrów) ciąg_instrukcji
```

Formalnie, dyrektywa **#define** nakazuje kompilatorami zamianę nazwy stałej na **ciąg_instrukcji**. Makrodefinicje mają liczne zastosowania. Np. często, przy wprowadzaniu danych z klawiatury na monitorze ukazuje się napis:

```
Podaj dane :
```

Wygodnie jest zastosować makrodefinicję:

```
#define DANE printf("Podaj dane")
```

Często też, podczas kończenia programu pojawia się napis:

```
Nacisnij ENTER aby skonczyc
```

W tym przypadku możemy zastosować makrodefinicję:

```
#define KONIEC printf("Nacisnij ENTER aby skonczyc")
```

Klasycznym przykładem jest makrodefinicja poszukiwania maksimum:

```
#define MAX(x,y) ((x)>(y)?(x):(y))
```

W przykładowym wywołaniu tej makrodefinicji:

```
liczba = MAX(m,n) ;
```

nastąpi substytucja tej instrukcji na:

```
liczba = ((m)>(n)?(m):(n));
```

Podczas projektowania makrodefinicji należy zwracać szczególna uwagę na poprawność językową - istnieje możliwość wystąpienia zastąpienia, które nie jest przez nas zamierzona, zaleca się stosowanie nawiasów.
Jako przykład stosowania makroinstrukcji pokażemy program, który (podobnie jak poprzedni) zamienia temperaturę podaną w stopniach Farenheita na temperaturę w stopniach Celsjusza.

Listing 8.12.Makrodefinicje

```
#include <stdio.h>
#include <conio.h>
#define ROZ 30
#define ZMIANA(t) (5.0/9.0)*(t-32)

int main()
{int i, n;
 int tem[ROZ];
 printf("\nPodaj ile danych : ");
 scanf("%d",&n);
 printf("\n");
 for (i = 0; i<n; i++)
   {printf("Podaj temperature [F]  ");
    scanf("%d",&tem[i]);
   }
 for (i = 0; i<n; i++)
    printf("\nTemperatura %d [F] = %.2f
              [C]",tem[i],ZMIANA(tem[i]));
 printf("\nNacisnij ENTER aby skonczyc");
 getche();
 return 0;
}
```

W następnym przykładzie ilustrujemy bardziej rozwiniętą makrodefinicję. Po uruchomieniu programu otrzymamy napis " TEKST W RAMCE " ograniczony od góry i dołu prostokątem, uzyskanym z rysowania znaku semigraficznego o

kodzie \xDB. Makrodefinicja ma postać:

```
#define NL printf("\n")
#define PODKR(n) NL;                    \
   for(i=1;i<(n);i++) printf("\xDB"); \
   NL;
```

Występujące znaki (\) oznaczają kontynuację makrodefinicji. Długość prostokątów określona jest zmienną **n.** Wywołania makrodefinicji mają postać:

```
PODKR(15)
```

Listing 8.13.Makrodefinicje

```
#include <stdio.h>
#include <conio.h>
#define NL printf("\n")
#define PODKR(n) NL;                    \
   for(i=1;i<(n);i++) printf("\xDB"); \
   NL;
void main(void)
{int i;
 PODKR(15)
 printf("TEKST W RAMCE");
 PODKR(15)
 getche();
}
```

Ważnym zagadnieniem jest poprawne projektowanie makrodefinicji. Rozważymy następujący przykład. Niech makrodefinicja ma postać:

```
#define SUMA(x,y) x+y
```

Tak zaprojektowaną makrodefinicję wywołamy w następujący sposób:

```
wynik = 10 * SUMA(1,2);
```

Po wykonaniu, prawdopodobnie spodziewamy się **wyniku** postaci:

```
wynik = 10 * 3 = 30
```

Pamiętamy jednak, że makra zamieniają tekst, w efekcie substytucja ma postać:

```
wynik = 10 * 1 + 2 = 12
```

Poprawna postać przykładowego makra powinna mieć postać:

```
#define SUMA(x,y) (x + y)
```

Widzimy, że makrodefinicje zachowują się jak funkcje. Powstaje pytanie - kiedy stosować funkcje a kiedy makrodefinicje. Należy pamiętać o ogólnej zasadzie:

*Makrodefinicja powoduje zwiększenie kodu,*
*za to wykonuje się szybciej niż funkcja.*

## 8.11. Funkcje rozwijalne (inline)

***Funkcje rozwijalne*** działają podobnie jak makrodefinicje. Funkcje ***inline*** nie są przyjęte w standardzie języka C (ma to miejsce dla języka C++), ale często do programowania w języku C używamy kompilatora języka C++ i wobec tego zdecydowaliśmy się na krótki opis tych funkcji. Zasadnicza różnica polega na tym, że kompilator traktuje funkcje rozwijalne jak zwykłe funkcje. W zasadzie, wprowadzenie do programy funkcji rozwijalnych jest jedynie zleceniem, aby kompilator starał się w każdym wywołaniu funkcji rozwijalnej zastąpić ją instrukcjami, które są w niej umieszczone. Ta sugestia nie będzie zrealizowana, jeżeli funkcja rozwijalna:

- jest funkcją rekurencyjną
- zawiera pętlę, instrukcję **switch**
- jest wywoływane przed jej definicją
- zawiera zbyt wiele instrukcji

Konstrukcja funkcji rozwijalnej ma postać:

```
inline typ nazwa_funkcji(lista_parametrów)
{   ciało funkcji
}
```

Klasyczne przykłady to:

a)

```
inline int abs(int i)   //wartość absolutna
   { return (i<0 ? -i : i);
   }
```

b)

```
inline int min(int a, int b)  //minimum
   { return (a<b ? a : b);
   }
```

c)

```
inline int parzysta(int i)
   { return (!(i%2));
   }
```

d)

```
inline int iloczyn(int a, int b)
   { return (a*b);
   }
```

Program ilustrujący stosowanie funkcji rozwijalnych może mieć postać pokazaną na listingu 8.14.

Listing 8.14.Funkcje *in line*

```
#include <stdio.h>
#include <conio.h>
#define KONIEC printf("\n nacisnij ENTER");

inline int parzysta (int x)
       { return(!(x%2));
       }
void main(void)
{int a;
 printf("\n podaj liczbe : ");
 scanf("%d",&a);
 if (parzysta(a))
     printf("\nliczba jest parzysta");
 else
     printf("\nliczba jest nieparzysta");
 KONIEC
 getche();
}
```

Program wymaga wprowadzenia liczby, następnie określa, czy liczba jest parzysta, czy nie.

## 8.12. Funkcje rekurencyjne

Funkcje, które wywołują same siebie noszą nazwę funkcji rekurencyjnych. Proces ten nosi nazwę rekurencji ( ang. *recursion*). Dyskusja, czy stosować funkcje rekurencyjne czy nie (programy rekurencyjne, zasadniczo można zastąpić programami iteracyjnymi) ciągle jest prowadzona. Stosując funkcje rekurencyjne zawsze musimy zastanowić się nad wprowadzeniem warunku stopu (łatwo zakodować program, który nigdy się nie zatrzyma!). Pojecie funkcji rekurencyjnej ilustruje się programami obliczającymi silnię.

Silnia oznaczana jako **n!** nieujemnej liczby całkowitej **n** jest iloczynem:

```
n(n-1)(n-2)(n-3).........1
```

Przyjmuje się, że 0! = 1 i 1! = 1. Dla przykładu:

```
5! = 5x4x3x2x1 = 120
```

Silnia liczby całkowitej **n** może być liczona iteracyjnie (wykorzystując np. pętlę **for**). Program pokazany na listingu 8.15 oblicza funkcję silnia iteracyjnie. Należy pamiętać, że silnia jest szybko wzrastającą funkcją, tak, że, nawet dla typu danych **unsigned long** argument **n** funkcji nie może przekraczać 12, aby nie przekroczyć limitu na największą możliwą liczbę całkowitą.

Listing 8.15.Obliczanie silni - iteracyjnie

```
#include <stdio.h>
#include <conio.h>
unsigned long silnia( unsigned long );
int main()
{ for ( int i = 0; i <= 12; i++)
  printf("%2d ! = %ld\n",i,silnia(i));
  getch();
  return 0;
}
unsigned long silnia (unsigned long num)
 { unsigned long wal = 1;
   for (int n =num; n >= 1; n--)
        wal *= n;
   return wal;
 }
```

Specyfikacja języka C wymaga, aby zmienna typu **unsigned long int** była przechowywana w przynajmniej 4 bajtach i w ten sposób mogła przechowywać wartości w zakresie od 0 do 4 294 967 295.

Rekurencyjna definicja funkcji silnia wyraża się wzorem:

```
n! = n(n-1)!
```

Rozwijając ten wzór, otrzymujemy np., że 5! równa się 5*4!. Program do obliczenia silni rekurencyjnie pokazany jest na kolejnym listingu.

Listing 8.16. obliczanie silni rekurencyjnie

```
#include <stdio.h>
#include <conio.h>
unsigned long silnia( unsigned long );
int main()
{
  for ( int i = 0; i <= 12; i++)
  printf("%2d ! = %ld\n",i,silnia(i));
  getch();
  return 0;
}

 unsigned long silnia (unsigned long num)
 {
  if (num <= 1) return 1;
  else return num* silnia(num-1);
 }
```

W wyniku wykonania programów z listingów 8.15 i 8.16 otrzymamy następujący wydruk:

```
0 ! = 1
1 ! = 1
2 ! = 2
3 ! = 6
4 ! = 24
5 ! = 120
6 ! = 720
7 ! = 5040
8 ! = 40320
9 ! = 362880
10 ! = 3628800
11 ! = 39916800
12 ! = 479001600
```

## 8.13. Argumenty funkcji main()

Starsze systemy operacyjne (Unix, DOS) uruchamiały programy z wiersza poleceń. Programy pisane w języki C/C++ mogą być uruchamiane z ***wiersza poleceń***. Wiersz poleceń (ang. *command line*) jest wierszem widocznym na ekranie monitora. W tym wierszu można wpisać odpowiedni tekst, który spowoduje uruchomienie programu. Konkretny program można uruchamiać z parametrami formalnymi. Parametry formalne można wpisać z wiersza poleceń. Funkcja **main(),** od której rozpoczyna się wykonywanie programu, posiada parametry formalne. W praktyce program uruchamiamy pisząc w linii poleceń nazwę programu i ewentualne parametry formalne. Kompilatory języka C zgodne ze standardem ASCII pozwalają, aby funkcja **main()** nie przyjmowała żadnych argumentów albo przyjmowała dokładnie dwa argumenty. Kompilator Borland Turbo C++ v.3.1 umożliwia korzystanie z trzech argumentów.

Standardowo dwa argumenty przekazywane funkcji **main()** nazywane są tradycyjnie **argc** i **argv**. Zmienna **argc** podaje ilość argumentów wprowadzonych z wiersza poleceń, Tablica **argv** jest tablicą wskaźników do typu **char**. Traktujemy ją, jako tablicę łańcuchów. Ponieważ element **argv[0]** zawiera zawsze nazwę programu, wartość **argc** będzie zawsze wynosiła, co najmniej 1. Na kolejnym listingu pokazano sposób przekazania argumentów do funkcji **main().**

Należy pamiętać, że zintegrowane środowiska programistyczne dla systemu Windows, takie jak Borland C/C++ czy Microsoft Visual C++ nie wykorzystują do uruchamiania programów wiersza poleceń (korzystamy z graficznych interfejsów). Z różnych jednak względów, niektóre z nich posiadają opcje, pozwalające na przekazanie argumentów do funkcji **main().** W środowisku programistycznym Borland Builder 5 należy utworzyć nową aplikacje tekstową a następnie skompilować, (ale nie wykonywać) wydając polecenie Project | Bulid All Projects. Następnie należy wybrać z menu głównego polecenie Run | Parameters. Dzięki tej opcji mamy możliwość podania parametrów, które będą

przekazane do funkcji **main().** Program pokazany na listingu 8.17 ilustruje przekazywanie argumentów do funkcji main().

Listing 8.17. Argumenty funkcji main()

```
#include <stdio.h>
#include <conio.h>
int main(int argc, char* argv[])
{ printf("Liczba argumentow = %d \n", argc);
  for (int i = 0; i < argc; i++)
    printf("Argument %d : %s\n",i,argv[i]);
  return 0;
}
```

W okienku dialogowym *Run Parameters* wpisujemy przykładowo trzy imiona:

```
Anna   Ela  Zofia
```

Po uruchomieniu programu otrzymamy następujący wydruk:

```
Liczba argumentow = 4
Argument 0 : C:\Program Files\Borland\CBuilder5\
             Projects\Project1.exe
Argument 1 : Anna
Argument 2 : Ela
Argument 3 : Zofia
```

W programie zadeklarowana funkcja **main():**

```
int main(int argc, char* argv[])
```

pobiera dwa argumenty i zwraca pojedynczą wartość. Z drugiej strony wiemy, że przekazywanie argumentów do funkcji następuje w momencie jej wywołania. Ale widzimy także, że funkcja **main()** nie jest jawnie nigdzie wywoływana. Funkcję **main()** wywołuje system operacyjny, zaś argumenty pobierane są z wiersza poleceń.

W większości przypadków wartość zwracana przez funkcje **main()** jest niewykorzystana, możemy nie umieszczać instrukcji **return** w ciele tej funkcji. Legalne deklaracje funkcji **main()** mogą mieć postać:

```
main();
int main();
int main(void);
int main(int argc, char* argv[]);
int main(int argc, char** argv);
void main();
void main(void);
void main(int argc, char* argv[]);
void main(int argc, char** argv);
```

Oczywiście argumenty przekazywane funkcji **main()** mogą być wykorzystane w programie.

Kolejny program pobiera trzy imiona, jako argumenty funkcji **main(),** oblicza długość każdego imienia i wyświetla imię tyle razy ile liter zawiera imię.

Listing 8.18. Argumenty funkcji main()

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

int main(int argc, char* argv[])
{ int k;
  for (int i = 1; i < argc; i++)
   {
     k = strlen(*(argv + i ));
     for (int j=1; j<=k; j++)
       printf("\n %s ",argv[i]);
    }
  return 0;
}
```

Jeżeli naszymi argumentami były łańcuchy:

```
Anna Ela
```

To otrzymamy następujący wynik:

```
Anna
Anna
Anna
Anna
Ela
Ela
Ela
```

W zasadniczym fragmencie programu:

```
k = strlen(*(argv + i ));
for (int j=1; j<=k; j++)
  printf("\n %s ",argv[i]);
```

korzystając z funkcji łańcuchowej **strlen(),** znajdującej się w pliku **<string.h>** ustalamy długość każdego łańcucha. W linii:

```
k = strlen(*(argv + i ));
```

wykorzystano fakt, że tablica może być traktowana, jako wskaźnik oraz, że do jej obsługi może być wykorzystana arytmetyka wskaźnikowa.
Zapis:

```
*(argv + i)
```

wskazuje na element **argv[i].** W tej sytuacji **argv[1]** reprezentuje łańcuch **Anna** i k = 4, a **argv[2]** reprezentuje łańcuch **Ela** i k = 3.

Argumenty odczytywane z wiersza poleceń są odczytywane, jako łańcuchy. Bardzo często chcemy z wiersza poleceń wprowadzić także wartości numeryczne. W takim przypadku należy skorzystać z funkcji **atoi()** (ang. *alphanumeric to integer*). Funkcja **atoi()** znajduje się w pliku **<stdlib.h>,** pobiera ona, jako argument łańcuch znakowy, a zwraca odpowiadającą mu wartość całkowitą. W kolejnym przykładzie argumenty funkcji **main()** mają postać:

```
Anna 3
```

Chcemy, aby imię Anna było powtórzone trzy razy. Program może mieć postać pokazaną na kolejnym listingu.

Listing 8.19. Numeryczne argumenty funkcji main()

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

int main(int argc, char* argv[])
{ for (int i = 1; i <= atoi(argv[2]); i++)
          printf("\n %s ",argv[1]);
  return 0;
}
```

Po uruchomieniu programu mamy następujący wydruk:

```
Anna
Anna
Anna
```

# ROZDZIAŁ 9

## TABLICE

## 9.1. Wstęp

Tablice pozwalają na wygodne operowanie zbiorami danych. Technicznie tablica jest wydzielonym jednorodnym obszarem pamięci. Ze swojej natury dane w pamięci komputera zapisane są sekwencyjnie. Podstawową konstrukcją jest tablica jednowymiarowa. Dla wygody użytkowników, twórcy języka C wprowadzili tablice wielowymiarowe. Należy pamiętać, że jest to jedynie konstrukcja logiczna. Bardzo często, aby przyśpieszyć wykonywanie operacji na tablicach wielowymiarowych, praktycy przekształcają takie tablice na jednowymiarowe. Z tablic wielowymiarowych największe znaczenie mają tablice dwuwymiarowe (tak zapisywane są obrazy cyfrowe), tablice o trzech i większych rozmiarach praktycznie są rzadko używane.

Tablica jest zbiorem danych, elementy zbioru są tego samego typu. Formalnie tablica jest jednorodną strukturą danych. Jedną z zalet stosowania tablic jest fakt, że danym tego samego typu nadajemy ten sam identyfikator. Tablicę możemy traktować, jako prostą zmienną z indeksem. Nawiasy kwadratowe - [ ] zawierają indeks. Tablice mogą być jednowymiarowe lub wielowymiarowe.

Jeżeli zapisujemy w ciągu 7 dni temperatury, to te dane może reprezentować tablica o nazwie **temperatura**. Przypuśćmy, że mamy następująca tabelę:

| Nr dnia | Temperatura [ $^{o}$C] |
|---|---|
| 1 | 21 |
| 2 | 24 |
| 3 | 25 |
| 4 | 20 |
| 5 | 18 |
| 6 | 23 |
| 7 | 22 |

Deklaracja tablicy w programie ma następująca postać:

```
int temperatura[7];
```

Tablica została zdefiniowana, jako zmienna typu **int** (kompilator musi znać typ, aby poprawnie zarezerwował pamięć). W nawiasie kwadratowym umieszczona jest liczba 7. Liczba w nawiasie kwadratowym informuje ile zmiennych typu **int** mieści się w tablicy o nazwie **temperatura**. Indeksowanie tablicy zawsze zaczyna się od 0. W nawiasie kwadratowym umieszczone może być oczywiście wyrażenie. Wartość wyrażenia musi być dodatnia, jest to rozmiar tablicy. Elementy przykładowej tablicy o nazwie **temperatura** mogą być następujące:

```
temperatura[0] = 21
temperatura[1] = 24
temperatura[2] = 25
temperatura[3] = 20
```

```
temperatura[4] = 18
temperatura[5] = 23
temperatura[6] = 22
```

Indeksy tablicy przebiegają wartości od zero do **sizeof - 1**. Maksymalnym indeksem w naszym przypadku jest liczba 6. Przy definiowaniu tablicy jest dobrym zwyczajem stosowanie symbolicznej stałej przy ustalaniu wymiaru tablicy:

```
#define WYMIAR 7
int temperatura[WYMIAR]  // zakres indeksów od 0 do 6
```

Jeżeli chcemy znać temperaturę zapisaną w trzecim dniu to wtedy odwołujemy się do odpowiedniego elementu tablicy, w naszym przypadku w następujący sposób:

```
temperatura[2]
```

## 9.2. Deklarowanie tablicy

Formalnie deklarujemy tablicę w następujący sposób:

Tablica jednowymiarowa:

```
typ_danych   nazwa_tablicy [rozmiar] =
{lista_wartości}
```

Tablica wielowymiarowa:

```
typ_danych   nazwa_tablicy [lista_rozmiarów] =
{lista_wartości}
```

**typ_danych** jest typem podstawowym lub zdefiniowanym,
**nazwa_tablicy** jest nazwą , taka sama jak dla zwykłych zmiennych,
**rozmiar** określa rozmiar tablicy (liczbę elementów) oraz zakres indeksowania,
**lista_rozmiarów** określa rozmiar tablicy wielowymiarowej i zakresy indeksowania,
**lista_wartości** jest zbiorem danych początkowych tablicy (inicjalizacja tablicy).

Elementy tablicy mogą być typu **int, double, float, short int, long int, char, wskaźnikowego, struct, union,** mogą być również tablicami.
Wartość rozmiaru musi być wyrażeniem możliwym do obliczenia w fazie kompilacji, zatem nie można używać zmiennej do określenia rozmiaru tablicy.

## 9.3. Inicjalizowanie tablic

Tablice należeć mogą do klasy **automatic**, **static** i **external**, ale nie do klasy **register**.

Zainicjalizowanie tablic możemy wykonać na kilka sposobów.

1. za pomocą deklaracji definiującej:

```
int tablica[5] = {2, 3, 4, 5, 6 };
int tablica[ ] = {2,3,4}
```

   jeżeli pominięto rozmiar tablicy, to kompilator obliczy rozmiar na podstawie ilości wprowadzonych danych, umieszczonych pomiędzy nawiasami klamrowymi,
2. deklarując tablicę jako **static**. Jeżeli nie poda się wartości inicjalizujących, kompilator zainicjalizuje tablice zerami.
3. Deklarując tablicę globalną (umieszczoną na zewnątrz wszystkich funkcji). Jeżeli nie poda się wartości inicjalizujących, kompilator zainicjalizuje tablice zerami.

Jeżeli tablica (przy deklaracji zmiennej automatycznej) nie zostanie zainicjalizowana, to odwołanie się do elementu tablicy jest błędem - pojawi się przypadkowa wartość. Zatem, gdy nie chcemy inicjalizować tablicy, powinniśmy zadeklarować tablicę, jako zmienną zewnętrzną lub statyczną. Sposób zastosowanie tablicy pokazuje kolejny program. W programie tablica **x** wypełniana jest kolejnymi liczbami całkowitymi i obliczana jest suma elementów tablicy.

Listing 9.1Tablice

```
/* tablica jednowymiarowa */
#include <stdio.h>
#include <conio.h>
#define ROZ 20
#define KOL 5
int main()
{  int k, x[ROZ];
   int suma = 0;
   for (k = 0; k<ROZ; ++k)
      {
       x[k] = k;
       suma += x[k];
      }
   printf("\n     Lista elementow");
   for (k=0; k <ROZ; ++k)
   printf("%c%4d",(k % KOL == 0)? '\n' : '\xB0',x[k]);
   printf("\nsuma elemntow tablicy = %d",suma);
   getche();
  return 0;
}
```

W dyrektywie preprocesora:

```
#define ROZ 20
```

zdefiniowaliśmy symbol **ROZ**, który ustala liczbę wartości w tablicy, dzięki temu możemy mieć deklarację tablicy **x** zapisaną, jako:

```
int k, x[ROZ];
```

**ROZ** jest wyrażeniem stałym, nie można zdefiniować **ROZ,** jako:

```
const int ROZ = 20      // błąd !
```

ponieważ **ROZ** w tym przypadku będzie stałą symboliczną.
W pętli **for**:

```
for (k = 0; k<ROZ; ++k)
         {    x[k] = k;
     suma += x[k];
       }
```

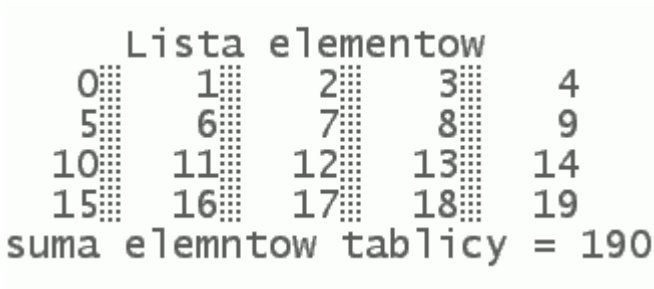
tablica **x** jest wypełniana wartościami, dzięki przypisaniu:

```
x[k] = k;
```

a następnie obliczana jest suma kolejnych elementów tablicy.
Fragment programu:

```
printf("\n     Lista elementow");
for (k=0; k <ROZ; ++k)
   printf("%c%4d",(k % KOL == 0)?'\n':'\xB0',x[k]);
```

powoduje wydruk listy elementów, w funkcji **printf()** zastosowaliśmy operator warunkowy, dzięki temu, mamy wydruk pięciu elementów w jednej linii.




Rys.9.1. Wynik działania programu

Możemy zmieniać liczbę elementów wydrukowanych w jednej linii, zmieniając dyrektywę preprocesora:

```
#define KOL 5
```

Na przykład zapis:

```
#define KOL 10
```

spowoduje wydruk dziesięciu elementów w jednej linijce.
Kolejny program, pokazany na listingu 9.2 wykonuje następujące zadanie.

Dana jest tablica **x** (wszystkie elementy są zerami) oraz tablica **y**, której elementami są liczby rzeczywiste. Program wybiera z tablicy **y** dodatnie liczy rzeczywiste i umieszcza je kolejno w tablicy **x**. Inicjalizacja tablic **x** i **y** zdefiniowana jest w następujących liniach:

```
float x[ROZ1]={0.0};
float y[ROZ2]={0.0,-1.0,0.0,3.0,6.0,-1.5,12.0,11.0,-
2.5,13.0};
```

Generalnie, jeżeli rozmiar tabeli jest **ROZ1** (w naszym przypadku ROZ1 = 10) to w nawiasie klamrowym powinno znajdować się dziesięć wartości. Jeżeli jest tych wartości mniej, kompilator automatycznie pozostałym elementom tablicy przypisuje zera. Tablica **y** zainicjalizowana jest standardowo.

Listing 9.2.Dwie tablice

```
#include <stdio.h>
#include <conio.h>
#define ROZ1 10
#define ROZ2 10
int main()
{  int i,j;
   float x[ROZ1]={0.0};
   float y[ROZ2]={0.0,-1.0,0.0,3.0,6.0,
                  - 1.5,12.0,11.0,-2.5,13.0};
   for (i=0, j=0; j<ROZ2; ++j)
      if (y[j] > 0)  x[i++] = y[j];
   for (j=i; j <ROZ2; j++)   x[j] = 0 ;
   printf("\n  tablica y \n");
   for (i=0; i<ROZ2;++i)         printf("%6.1f",y[i]);
   printf("\n Nowa lista elementow \n");
   for (i=0; i<ROZ1;++i)         printf("%6.1f",x[i]);
   getche();
   return 0;
}
```

Główne zadanie wykonane jest w pętli **for**:

```
for (i=0, j=0; j<ROZ2; ++j)
     if (y[j] > 0)  x[i++] = y[j];
for (j=i; j <ROZ2; j++)   x[j] = 0 ;
```

Dodatnie elementy tablicy **y** są kopiowane do tablicy **x** (pierwsza pętla **for**), druga pętla **for** powoduje przypisanie zer pozostałym elementom tablicy **x**. Na końcu drukowana jest tablica **y** oraz zmodyfikowana tablica **x**.
Po uruchomieniu wynik działania programu ma postać:

```
Tablica y
0.0   -1.0    0.0     3.0     6.0   -1.5   12.0   11.0
-2.5   13.0
```

```
Nowa lista elementow
3.0    6.0   12.0   11.0   13.0    0.0     0.0    0.0
0.0     0.0
```

Tablice są bardzo często wykorzystywane do przechowywania wartości funkcji matematycznych. Następny program (9.3) demonstruje produkowanie tabeli wartości dla rozkładu normalnego oraz obliczanie pola powierzchni pod krzywą

Listing 9.3. Krzywa Gaussa

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define ROZ 100
int main()
{float z,z1,z2,dz, sq, y[ROZ], pole, suma;
 int n,i,k,a;
 printf("\n n <= %d,",ROZ);
 printf("  podaj z1, z2, n :  ");
 scanf("%f%f%d",&z1,&z2,&n);
 dz =  (z2-z1)/n;
 z = z1;    i = 0;    suma = 0.0;
 sq = 1.0/sqrt(2.0*3.1415926);
 do
  { y[i] = exp(-z*z*0.5)*sq;   //krzywa Gaussa
    suma += y[i];
    a = 150*y[i];              //wykres
    printf("%6.3f",z);
    for (k=0;k<=a;k++)
    printf("\xCD");
    printf("\n");             //koniec wykresu
    z += dz;
    i++;
  } while (z < z2 + dz*0.5 );
  pole = dz*suma;           //pole pod krzywa Gaussa
  printf("Powierzchnia pod krzywa Gausa=9.7f",pole);
  printf("   nacisnij ENTER");   getch();
  clrscr();
  printf("\n    z         y       dz = %5.4f\n",dz);
  for (k=0; k<i ;k++)
    printf("\n%10.5f%10.5f",z1+k*dz,y[k]);
  return 0;
}
```

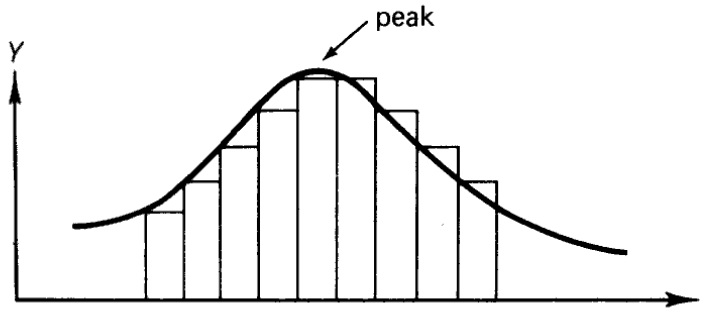
Funkcja opisująca rozkład normalny ma postać (tzw. **krzywa normalna**):

$$Y = \frac{e^{-Z^2/2}}{\sqrt{2\pi}}$$

W badaniach statystycznych potrzebujemy czasem obliczyć powierzchnię pod krzywą normalną. Można to zrobić na wiele sposobów. My zastosujemy prostą metodę. Powierzchnię pod krzywą dzielimy na prostokąty, wysokość prostokąta jest $Y_n$ a szerokość dz.
Pole powierzchni S jest równe:

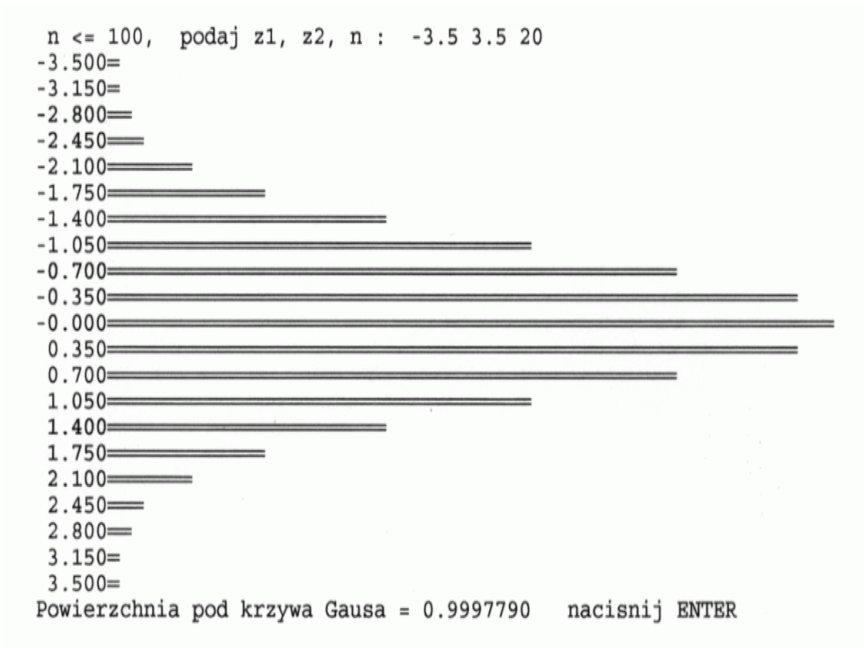
$$S = Y_1dz + Y_2dz + Y_3dz + .....+ Y_ndz = dz(Y_1 + Y_2 + Y_3 + .....+ Y_n)$$

Mamy do czynienia z całkowaniem numerycznym. Dokładność całkowania zależy oczywiście od szerokości prostokątów - im mniejsza szerokość tym dokładniejszy wynik, (ale tym samym musimy mieć więcej prostokątów i wobec tego wzrośnie czas obliczeń). Dokładna wartość całki (w granicach od $-\infty$ do $+\infty$ ) jest 1. Zasada całkowania pokazana jest na rysunku 9.2, pokazano technikę całkowania numerycznego metoda prostokątów.




Rys. 9.2. Pole pod krzywą jest obliczone w przybliżony sposób

Po uruchomieniu programu, na ekranie pojawia się wykres funkcji (Rys.9.3) i wydruk wartości funkcji Gaussa (Rys. 9.4).




Rys.9.3 Wykres funkcji Gaussa (ekran nr 1)

| z | y | dz = 0.3500 |
|---|---|---|
| -3.50000 | 0.00087 | |
| -3.15000 | 0.00279 | |
| -2.80000 | 0.00792 | |
| -2.45000 | 0.01984 | |
| -2.10000 | 0.04398 | |
| -1.75000 | 0.08628 | |
| -1.40000 | 0.14973 | |
| -1.05000 | 0.22988 | |
| -0.70000 | 0.31225 | |
| -0.35000 | 0.37524 | |
| -0.00000 | 0.39894 | |
| 0.35000 | 0.37524 | |
| 0.70000 | 0.31225 | |
| 1.05000 | 0.22988 | |
| 1.40000 | 0.14973 | |
| 1.75000 | 0.08628 | |
| 2.10000 | 0.04398 | |
| 2.45000 | 0.01984 | |
| 2.80000 | 0.00792 | |
| 3.15000 | 0.00279 | |
| 3.50000 | 0.00087 | |

Rys.9.4 Tabela wartości funkcji Gaussa

Program tworzy tablicę wartości funkcji i umieszcza je w tablicy **y.** Tablica **y** deklarowana jest następująco:

```
#define ROZ 100
float z,z1,z2,dz, sq, y[ROZ], pole, suma;
```

Rozmiar tablicy **y** możemy zmienić definiując na nowo dyrektywę preprocesora. Program żąda wprowadzenia trzech zmiennych: dolnej wartości zakresu zmienności **z1**, górnej wartości zmienności **z2** oraz liczby obliczanych wartości **n**. Następnie ustala krok obliczeń **dz:**

```
dz =  (z2-z1)/n;
```

Obliczanie wartości funkcji oraz niezbędną sumę (zmienna **suma**) potrzebną do obliczenia pola pod krzywą realizuje następujący fragment programu:

```
do
  { y[i] = exp(-z*z*0.5)*sq;    //krzywa Gaussa
    suma += y[i];
    a = 150*y[i];               //wykres
    printf("%6.3f",z);
    for (k=0;k<=a;k++)
       printf("\xCD");
    printf("\n");              //koniec wykresu
    z += dz;
    i++;
  } while (z < z2 + dz*0.5 );
```

W tym fragmencie tworzony jest także wykres funkcji:

```
a = 150*y[i];               //wykres
printf("%6.3f",z);
for (k=0;k<=a;k++)
    printf("\xCD");
```

W jednym przebiegu pętli **do...while** drukowana jest jedna linia. Wartość funkcji (maksymalna wartość wynosi 0.3989) została pomnożona przez liczbę 150. Jest to potrzebne gdyż wykres tworzony jest w **trybie tekstowym.** Funkcja **printf()** umieszcza na ekranie znak semigraficzny (o kodzie **xCD** ). Ilość tych znaków jest proporcjonalna do wartości funkcji. Zakładamy, że na ekranie umieścić możemy 80 znaków. Tajemniczy warunek w pętli **do...while**:

```
while (z < z2 + dz*0.5 );
```

spowodowany jest faktem, że pewne wartości nie mogą być reprezentowane dokładnie w systemie binarnym (takim przykładem jest liczba 0.2, jej reprezentacja w systemie binarnym ma postać 0.1999999...). Dlatego nie możemy stosować testu "mniejszy, równy", ale poprawnie: "mniejszy". Być może, zechcemy wyniki skierować do drukarki. Następny program może być w tym pomocny. Jest to zmodyfikowany poprzedni program.

Listing 9.4 .Krzywa Gaussa, wykorzystanie drukarki

```
/* tablice, drukarka*/
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define ROZ 100
int main()
{float z,z1,z2,dz;
 FILE *f;
 float sq, y[ROZ], pole, suma;
 int n,i,k,a;
 if((f=fopen("prn","w"))==NULL)
 {
  printf("\nBlad otwarcia drukarki!");
  return;
 }
 printf("\n n <= %d,",ROZ);
 printf("  podaj z1, z2, n :  ");
 scanf("%f%f%d",&z1,&z2,&n);
 dz =  (z2-z1)/n;
 z = z1;
 i = 0; suma = 0.0;
 sq = 1.0/sqrt(2.0*3.1415926);
 fprintf(f,"\n  krzywa Gaussa (%4.2f,%4.2f)\n",z1,z2);
 do
  {
    y[i] = exp(-z*z*0.5)*sq;   //krzywa Gaussa
    suma += y[i];
    a = 150*y[i];              //wykres
     fprintf(f,"%6.3f",z);
    for (k=0;k<=a;k++)
     fprintf(f,"\xCD");
     fprintf(f,"\n");          //koniec wykresu
    z += dz;
    i++;
  } while (z < z2 + dz*0.5 );
    pole = dz*suma;           //pole pod krzywa Gaussa
    fprintf(f,"\nPowierzchnia pod krzywa Gausa
                = %9.7f \n",pole);
    fprintf(f,"\n    z       y    dz = %5.4f\n",dz);
    for (k=0; k<i ;k++)
     fprintf(f,"\n%10.5f%10.5f",z1+k*dz,y[k]);
  fclose(f);
  return 0;
}
```

# ROZDZIAŁ 10

## WSKAŹNIKI

## 10.1. Wstęp

Uproszczona definicja mówi, że wskaźnik jest zmienną, której wartością jest adres innej zmiennej. Wskaźniki są ważnym elementem w języku C, (kto wie, czy nie najważniejszym). Stosowanie wskaźników pozwala na pisanie wydajnych i szybkich programów. Należy zwrócić uwagę, że opanowanie poprawnego posługiwania się wskaźnikami jest podstawą eleganckiego i wydajnego programowania. Twórcy języka C dostarczyli programistom potężne narzędzie, ale też należy przypomnieć, że wskaźniki są bardzo niebezpiecznym elementem języka. Drobne błędy w obsłudze wskaźników mogą całkowicie zawiesić program komputerowy a nierzadko mogą powodować inne szkody w systemie.

Język C pozwala na bardzo zaawansowane używanie wskaźników i arytmetyki wskaźnikowej. Szczególnie istotne znaczenie mają wskaźniki w operowaniu tablicami a także w przekazywani adresu zmiennej do funkcji.

Wiadomo, że zmienne używane w programie są przechowywane w określonym miejscu pamięci, lub inaczej mówiąc, pod określonym adresem. Wskaźniki umożliwiają dostęp do pamięci i manipulowanie adresami.

## 10.2. Deklaracja typu wskaźnikowego

Typ wskaźnikowy musi być zadeklarowany, a deklaracja ma postać:

```
typ_wskazywanej_zmiennej *nazwa_zmiennej_wskaźnikowej
```

Np. deklaracja:

```
int *p
```

mówi kompilatorowi, że została utworzona zmienna wskaźnikowa **p**, (co sygnalizuje gwiazdka) służąca do obsługiwania przechowywanej w pamięci danej typu **int**. Typ wskazywanej zmiennej może być dowolny. Napis:

```
double *suma
```

oznacza to, że utworzyliśmy wskaźnik o nazwie **suma**, który będzie zawierał adres zmiennej typu **double**. Wartościami zmiennej wskaźnikowej mogą być wskaźniki do zadeklarowanych już obiektów (zmiennych, stałych, itp.) typu wskazywanego. Przypisanie zmiennej wskaźnikowej wskaźnika 0 (**NULL**) powoduje, że zmienna ta nie wskazuje na żaden obiekt.

## 10.3. Adres i rozmiar zmiennej

Do zmiennej możemy odwołać się na dwa sposoby:

- Używając jej nazwy - wtedy odnosimy się do wartości przechowywanej w pamięci,
- Używając operatora pobrania adresu (operator **&**) - wtedy odwołujemy się do adresu w pamięci, pod którym przechowywana jest zmienna.

Podstawowe pojęcia związane ze stosowaniem wskaźników pokażemy na serii krótkich programów. W pierwszym programie systematyzujemy dotychczasowe wiadomości - klasyczne odwołanie się do zmiennej, określenie ilości bitów, jakie są potrzebne do przechowania konkretnego typu danych, adres w pamięci. Wykorzystamy dwa operatory:

- Operator rozmiaru - **sizeof** (nazwa_typu)
- Operator adresu - **&**

W programie pokazanym na listingu 10.1 zdefiniowane są cztery zmienne: dwie zmienne typu **int** (**x1,x2**) i dwie zmienne typu **float** (**y1,y2**). Program określi ilość bajtów potrzebnych do zapisu tych danych oraz poda adres.

Listing 10.1 Operatory rozmiaru i adresu

```
/* wskazniki */
#include <stdio.h>
#include <conio.h>
int main()
{int x1,x1s,y1s;
 int x2;
 float y1;
 float y2;
 x1 = 1;
 x2 = 10;
 y1 = 15.15;
 y2 = 74.85;
 x1s = sizeof(x1);
 y1s = sizeof(y1);
 clrscr();
 printf("\nzmienna x1 = %6d, adres = %8u, %2i  bajty",
         x1,&x1,x1s);
 printf("\nzmienna x2 = %6d, adres = %8u",x2,&x2);
 printf("\nzmienna y1 = %6.2f,adres = %8u, %2i bajty",
         y1,&y1,y1s);
 printf("\nzmienna y2 = %6.2f, adres = %8u",y2,&y2);
 getche();
 return 0;
}
```

Po uruchomieniu programu mamy wydruk:

```
zmienna x1 =     1,  adres = 65524,  2 bajty
zmienna x2 =    10,  adres = 65518
zmienna y1 = 15.15,  adres = 65514,  4 bajty
zmienna y2 = 74.85,  adres = 65510
```

Widzimy, że zmienne całkowite ***(x1 i x2***) mają rozmiar 2 bajtów (ta wartość zależy od konkretnego kompilatora i komputera) i umieszczone są pod adresami

65524 i 65518 (te adresy są przypadkowe, w różnych systemach mogą być inne, ważna jest sekwencja zapisów).

Do zapisu jednej zmiennej **x1** potrzebne są 2 bajty o adresach 65524 i 65523. Zmienna typu **float** potrzebuje 4 bajtów, pierwsza z nich (**y1**) jest umieszczona pod adresem 65514. W programie mamy w sumie zdefiniowane trzy zmienne typu **int** i dwie zmienne typu **float**, kompletny zestaw adresów (w naszym systemie) ma postać:

```
x1  = 65524
x1s = 65522
y1s = 65520
x2   = 65518
y1  = 65514
y2  = 65510
```

## 10.4. Odwołanie do zmiennej przez wskaźnik

Możemy do zmiennych odwoływać się przez wskaźniki. W tym celu musimy zadeklarować wskaźnik - (deklaracja wskaźnika jest taka sama jak deklaracja każdej innej zmiennej), z tym, że musimy dodatkowo zastosować operator wskaźnika ( znak gwiazdki *****). Operator wskaźnika nazywany jest czasem w literaturze ***operatorem wskazania pośredniego***. W kolejny programie wprowadzamy deklarację wskaźnika.

Listing 10.2.Deklaracja wskaźnika

```
#include <stdio.h>
#include <conio.h>
int main()
{ int x1;         //deklaracja zmiennej
  int *px1;      //deklaracja wskaznika
  x1 = 10;
  px1 = &x1;     //px1 wskazuje na x1
  printf("\n(odwolanie przez nazwe) zmienna x1 =
          %4d",x1);
  printf("\n(odwolanie przez adres) zmienna x1 =
          %4d",*px1);
  getche();
  return 0;
}
```

Po uruchomieniu tego programu mamy następujący wydruk:

```
(odwolanie przez nazwe)  zmienna x1 = 10
(odwolanie przez adres)  zmienna x1 = 10
```

W tym programie zadeklarowany jest wskaźnik o nazwie **px1**:

```
int *px1;      //deklaracja wskaznika
```

**px1** jest wskaźnikiem do zmiennej typu **int** (nie można mieszać wskaźników do

różnych typów). Aby zmiennej ***px1*** przypisać wartość (adres) musimy zastosować specjalny operator adresu (**&**):

```
px1 = &x1;     //px1 wskazuje na x1
```

Dla ułatwienia posługiwania się wskaźnikami możemy zapamiętać, że powyższą instrukcję czytamy:

*zmienna* **px1** *otrzymuje adres zmiennej* **x1,**
**px1** *wskazuje na* **x1**

Aby uzyskać dostęp do zmiennej **x1** możemy posłużyć się zmienną **px1**, ale musimy zastosować operator wskaźnika (znak *):

```
printf("\n(odwolanie przez adres) zmienna x1=%4d",*px1);
```

Dla ułatwienia posługiwania się odwołaniem przez adres, możemy zapamiętać, że zapis typu ***zmienna_wskaźnikowa** czytamy:

*wartość umieszczona pod tym adresem*

W naszym przypadku zapis ***px1** czytamy, jako:

*wartość pod adresem umieszczonym w zmiennej* **px1**

## 10.5. Przypisanie wskaźnika

Wskaźnikami operujemy tak jak innymi zmiennymi. Jeżeli zadeklarujemy dwa wskaźniki, to możemy podstawić jeden wskaźnik do drugiego. Ilustruje to następny program.

Listing 10.3 Dwa wskaźniki - przypisanie

```
/* wskazniki */
#include <stdio.h>
#include <conio.h>
int main()
{int x1;  //deklaracja zmiennej
 int *px1;         //deklaracja wskaznika
 int *rx1;          //deklaracja innego wskaznika
 x1 = 10;
 px1 = &x1;    //px1 wskazuje na x1
 rx1 = px1;     //podstawienie wskaznika
 printf("\n(odwolanie przez 1 adres) zmienna x1 =
          %4d",*px1);
 printf("\n(odwolanie przez 2 adres) zmienna x1 =
          %4d",*rx1);
 getche();
 return 0;
}
```

Po uruchomieniu tego programu mamy następujący wydruk:

```
(odwolanie przez 1 adres) zmienna x1 = 10
(odwolanie przez 2 adres) zmienna x1 = 10
```

W programie mamy dwie deklaracje:

```
int *px1;         //deklaracja wskaznika
int *rx1;         //deklaracja innego wskaznika
```

Podstawienie ma prostą postać:

```
rx1 = px1;       //podstawienie wskaźnika
```

## 10.6. Operacje arytmetyczne na wskaźnikach

Ważnym mechanizmem posługiwania się wskaźnikami jest możliwość wykonywania operacji arytmetycznych na wskaźnikach. Jeżeli zmienna **p** jest wskaźnikiem do konkretnego typu, wtedy w pełni dopuszczalne są wyrażenia takie jak np.:

```
p  - 1
p += i
++p
```

Zastosowania arytmetyki wskaźnikowej pokazany jest w programie 10.4.

Listing 10.4.Arytmetyka wskaźnikowa

```
#include <stdio.h>
#include <conio.h>
int main()
{int x1,x2;
 double y1,y2;
 int *px1, *px2;
 double *py1,*py2;
 x1 = 10;
 x2 = 333;
 y1 = 9.99;
 y2 = 999.999;
 px1 = &x1;
 py1 = &y1;
 px2 = px1 - 1;  //arytmetyka wskaznikowa
 py2 = py1 - 1;  //arytmetyka wskaznikowa
 printf("\n zmienna x1 = %10d, adres = %8u",*px1,&x1);
 printf("\n zmienna x2 = %10d, adres = %8u",*px2,&x2);
 printf("\nzmienna y1 = %10.4f,adres = %8u",*py1,&y1);
 printf("\nzmienna y2 = %10.4f,adres = %8u",*py2,&y2);
 getche();
 return 0;
}
```

W operacjach arytmetycznych, kompilator języka C posługuje się tzw. bazowym typem wskaźnika. Dla danej typu **int** wynosi on 2, dla typu **float** - 4. Dla zmiennych typu **int** zapis:

```
p - 1
```

oznacza "od wartości adresu odejmij liczbę dwa", a dla zmiennych typu **float** oznacza "od wartości adresu odejmij liczbę 4". Kompilator wykonuje to automatycznie.
Po uruchomieniu programu z listingu 10.4 mamy następujący wydruk:

```
zmienna  x1 =       10,  adres = 65524
zmienna  x2 =      333,  adres = 65522
zmienna  y1 =   9.9900,  adres = 65514
zmienna  y2 = 999.9990,  adres = 65506
```

Deklaracje wskaźników mają postać:

```
int *px1, *px2;
double *py1,*py2;
```

Operacje na wskaźnikach są następujące:

```
px2 = px1 - 1;  //arytmetyka wskaznikowa
py2 = py1 - 1;  //arytmetyka wskaznikowa
```

Podsumujemy teraz nasze rozważania. Jeżeli mamy deklarację:

```
int *wsk;
```

to możemy mieć następujące zapisy:

```
1. wsk = 0;
2. wsk = NULL;           //równoważne z (1)
3. wsk = &k;
4. wsk = (int *) 25550;  // absolutny adres w pamięci
```

Zapis trzeci możemy czytać, jako:

***wsk*** *odnosi się do* ***k***
***wsk*** wskazuje na **k**
***wsk*** zawiera adres **k**

W zapisie czwartym tzw. rzutowanie (**int***) jest potrzebne, aby kompilator nie wysyłał ostrzeżenia. Operator * (zwany także operatorem wskazania pośredniego) jest operatorem unarnym, działa "z prawej strony w lewo". Jeżeli **wsk** jest wskaźnikiem, to ***wsk** jest wartością zmiennej, która jest umieszczona pod adresem **wsk**. Termin "***wskazanie pośrednie***" pochodzi z języków programowania maszynowego. Wartością **wsk** jest pozycją w pamięci (adres), natomiast ***wsk** jest wartością pośrednią **wsk**, w tym sensie, że jest to wartość w miejscu pamięci przechowywanym w **wsk**. Można powiedzieć, że operator * działa odwrotnie do operatora **&**.

Jeżeli mamy deklarację:

```
float a,b,*wsk;
```

wtedy dwie instrukcje:

```
wsk = &a;
b = *wsk;
```

są równoważne instrukcji :

```
b = *&a;
```

a to z kolei jest równoważne instrukcji:

```
b = a;
```

## 10.7. Inicjalizowanie wskaźników

Możemy inicjalizować wskaźnik, można to wykonać następująco:

```
int k = 10,
wsk = &k;
```

Zauważmy, że zainicjowano **wsk**, a nie ***wsk**. Zmienna **wsk** jest typu **int *** a jej początkowa wartość jest **&k**.
Jeżeli mamy deklaracje:

```
int *wsk;
double a = 99.99;
```

to zapis:

```
wsk = &a; //błąd w tym kontekście
```

jest niedopuszczalny, ponieważ wskaźnik do danej typu **double** nie może być przypisany do wskaźnika do typu **int**.
Należy pamiętać o kilku ograniczeniach:

- nie wskazywać na stałą (np. **&7** jest nielegalne)
- nie wskazywać na tablicę (np. **int t[100]; &t** - nazwa tablicy jest stałą)
- nie wskazywać na zwykłe wyrażenia (np. &(x - 100) jest nielegalne)
- nie wskazywać na zmienną rejestrową (np. **register x**; **&x** jest nielegalne)

## 10.8. Funkcje i wskaźniki

Wszystkie argumenty przekazywane są do funkcji techniką "***przekazywanie przez wartość***" (ang. ***call by value***). Znaczy to, że pamiętane argumenty w funkcji **main()** nie mogą być zmienione z poziomu wołanej funkcji.

W języku C istnieje jeszcze jeden mechanizm przekazywania argumentu w funkcji - jest to tzw. odwołanie przez **referencje** (ang. ***call by reference***). W funkcji musimy zadeklarować parametr jako wskaźnik. Gdy adres jest przekazywany, jako argument, funkcja ma możliwość zmiany wartości adresowanej zmiennej w środowisku wywołującym.
Poniższy program przykładowy ilustruje "odwołanie przez referencję". Po wprowadzeniu dwóch liczb, program określa, która z nich jest większa.

Listing 10.5 Odwołanie przez referencję

```
/* funkcje i wskazniki */
#include <stdio.h>
#include <conio.h>

void max(int,int,int*);   // zmienna wskaźnikowa

int main()
{int x,y,z;
 printf("\n program wskazuje liczbe wieksza z liczb
         calkowitych");
 printf ("\npodaj 1 liczbe :");
 scanf("%d",&x);
 printf("\npodaj 2 liczbe :");
 scanf("%d",&y);
 max(x,y,&z);
 printf("\nwieksza liczba to : %d",z);
 getche();
 return 0;
} //.....koniec main() .......

void max(int a,int b,int *v)
 { *v = (a > b) ? a : b;
 }
```

W tym programie deklaracja funkcji ma postać:

```
void max(int,int,int*);
```

Definicja funkcji ma postać:

```
void max(int a,int b,int *v)
{   *v = (a > b) ? a : b;
}
```

Wywołanie tej funkcji ma postać:

```
max(x,y,&z);
```

Zmienna **z** będzie zawierać większą z liczb **a** i **b**. Stosując tą metodę, możemy przekazywać więcej wartości.

Jeżeli chcemy np. wydrukować dwie liczby w ustalonym porządku (np. wzrastającym) to wykona takie zadanie następująca funkcja:

```
fun(int *a, int *b)
{int temp;
   if (*a >*b)
    { temp = *a;
      *a = *b;
      *b = temp;
    }
}
```

Jeżeli tą funkcję wywołamy w następujący sposób:

```
fun(&x, &y) ;
```

to spowodujemy, że **x** i **y** będą umieszczone w ustalonym porządku. Należy zauważyć, że adresy **x** i **y** zostały przekazane "***przez wartość***" i tych adresów nie można zmienić w środowisku wywołującym za pomocą funkcja **fun().** Natomiast za pomocą **dereferencji** funkcja może zmienić pamiętane wartości **x** i **y** pod podanymi adresami.

Aby stosować odwołanie przez referencje należy postępować następująco:

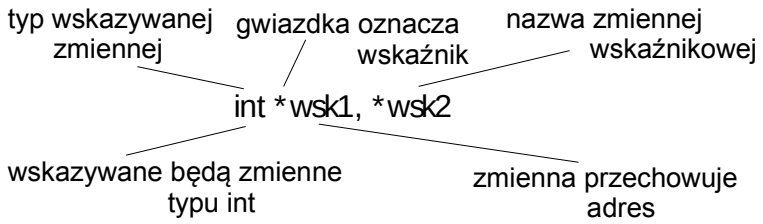
- parametr funkcji musi być zadeklarowany, jako wskaźnik
- wewnątrz ciała funkcji musi nastąpić dereferencja wskaźnika
- należy przekazać adres aktualny parametr funkcji

Stosowanie wskaźników jest dość skomplikowane, ale przynosi wiele korzyści:

- funkcja może zwrócić do środowiska wywołującego więcej niż jedną wartość.
- za pomocą wskaźników przekazywane są tablice i łańcuchy
- za pomocą wskaźników budowane są skomplikowane struktury danych
- za pomocą wskaźników można uzyskać użyteczne informacje techniczne (np. ilość wolnej pamięci).

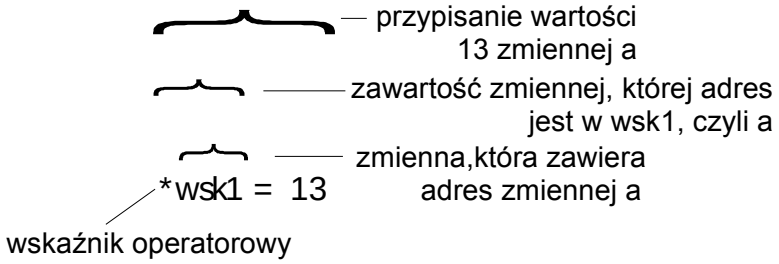
Należy zapamiętać, że:

*Stała wskaźnikowa jest adresem*
*Zmienna wskaźnikowa jest miejscem przechowującym adres*



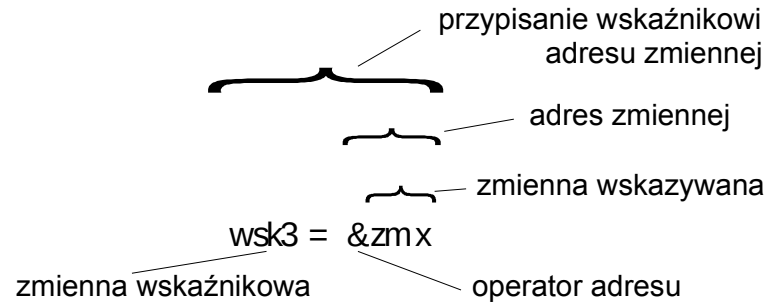

Rys. 10.1 Deklaracja zmiennej wskaźnikowej

Schematycznie deklaracja zmiennej wskaźnikowej pokazana jest na rysunku 5.5. Konstrukcja operatora wskaźnikowego pokazana jest na rysunku 10.2.




Rys. 10.2 Operator wskaźnikowy

Konstrukcja operatora adresu pokazana jest na rysunku 10.3.




Rys. 10.3 Operator adresu

Należy zapamiętać, że:

*Symbol * w definicji oznacza typ wskaźnikowy*
*Symbol * w instrukcji oznacza zmienną wskazywaną*

## 10.9. Tablice i wskaźniki

Tablice służą, do przechowywania jednorodnych danych, tablicę możemy traktować, jako prostą zmienną z indeksem. W języku C pomiędzy tablicami i wskaźnikami istnieje ścisły związek. Każda operacja na zmiennej indeksowanej może być wykonana także za pomocą wskaźników. W większości przypadków stosowanie wskaźników przyspiesza wykonanie instrukcji. W swojej istocie, zapis tablicowy jest ukrytym zapisem wskaźnikowym. Podczas kompilacji, tłumaczony jest zapis tablicowy na zapis wskaźnikowy. Nazwa tablicy jest automatycznie przekształcana na wskaźnik do pierwszego elementu tablicy. Nazwa tablicy jest adresem tablicy.

W następującym programie zilustrujemy zastosowanie wskaźników podczas operowania tablicą. Pokazany program wczytuje ustaloną przez użytkownika

ilość danych, umieszcza je w tablicy. Następnie wyszukuje największy i najmniejszy element tablicy.

Listing 10.6 Tablice i wskaźniki

```
/* tablice i wskazniki */
#include <stdio.h>
#include <conio.h>
#define ROZ 100
int main()
{int i, nl, min,max;
 int tab[ROZ];
 printf("\nprogram szuka maksymalnej i minimalnej
         liczby w tablicy\n ");
 printf("Ile liczb calkowitych :");
 scanf("%d",&nl);
 printf("\nwprowadz n = %d danych\n",nl);
 for (i=0; i<nl; i++)
 { printf("liczba nr %3d = ",i+1);
   scanf("%d",tab+i);      //uwaga na format !
 }
 max = min = *tab;
 for (i = 1; i<nl; i++)
     {if (*(tab + i) > max) max = *(tab+i);
      if (*(tab + i) < min) min = *(tab+i);
     }
 printf("\nmaksimum = %d , minimum = %d",max,min);
 getche();
 return 0;
}
```

Za pomocą dyrektywy preprocesora ustalamy rozmiar tablicy:

```
#define ROZ 100
```

a następnie deklarujemy tablicę:

```
int tab[ROZ];
```

Za pomocą pętli **for** wprowadzamy elementy tablicy:

```
for (i=0; i<nl; i++)
 {
   printf("liczba nr %3d = ",i+1);
   scanf("%d",tab+i);  //uwaga na format !
 }
```

Formalnie elementy tablicy **tab** mogą być wczytywane także w następujący sposób:

```
scanf("%d",&tab[i]);
```

Wyszukiwanie elementu największego i najmniejszego realizowane jest w

następującym fragmencie:

```
max = min = *tab;
 for (i = 1; i<nl; i++)
     {if (*(tab + i) > max) max = *(tab+i);
      if (*(tab + i) < min) min = *(tab+i);
     }
```

W zapisach:

```
max = *tab;
min = *tab;
```

zmiennym **max** i **min** przypisano wartość pierwszego elementu tablicy **tab** (pamiętamy, że nazwa tablicy jest wskaźnikiem do pierwszego elementu tablicy). Równoważne zapisy to:

```
max = tab[0];   //odwołanie przez indeks
min = tab[0];
```

W zapisie:

```
if (*(tab + i) > max) max = *(tab+i);
```

mamy wyrażenie:

```
*(tab + i)
```

Mamy równoważność zapisów:

```
*(tab + i)      jest równoważne     tab[i]
```

zapis ***(tab+i)** oznacza dostęp do elementu tablicy o nazwie **tab**, którego indeksem jest wartość zawarta w zmiennej **i.** Jeżeli np. i = 2, to otrzymujemy wartość elementu **tab[2].**

Należy zwrócić uwagę na interpretację zapisu ***(tab+i).** Wiemy, że **tab** jest adresem tablicy. Jeżeli dodamy do tego adresu liczbę np. 1 ( **i = 1**), to wyrażenie **tab + 1** da nam nowy adres. Nie będzie to adres zwiększony o jeden bajt (jak można by przypuszczać), ale adres następnego elementu tablicy **tab**. Kompilator zna typ zmiennej i dodaje potrzebną liczbę bajtów automatycznie (w większości przypadków dwa bajty dla zmiennej typu **int**). Należy zwrócić uwagę na pewne subtelne różnice w operowaniu tablicami. Rozpatrzymy fragment programu:

```
int tab[4] = {1, 2, 3, 4};
int *wsk;
wsk = tab;
```

Kolejno mamy:

- deklarację inicjującą tablicę **tab**
- deklaracja zmiennej wskaźnikowej **wsk**
- instrukcję przypisania

Instrukcja przypisania:

```
wsk = tab
```

jest równoważna instrukcji

```
wsk = &tab[0]
```

Zmienne **wsk** oraz **tab** wskazują na pierwszy element tablicy **tab**, czyli na element **tab[0**]. Różnica pomiędzy **wsk** i **tab** polega na tym, że identyfikator tablicy (u nas **tab**) jest inicjowany adresem jej pierwszego elementu, adres ten nie może ulegać zmianie w programie. Jest to wskaźnik stały i nie można go zwiększać ani zmniejszać. **Wsk** jest zmienną wskaźnikową, po ustawieniu wskaźnika na adres tablicy, możemy go zmieniać. Zapis:

```
tab = tab + 1  jest błędny
```

natomiast zapis :

```
wsk = wsk + 1    jest poprawny
```

W następnym programie wykorzystamy możliwość inkrementacji wartości zmiennych wskaźnikowych. Program ma za zadanie podąć wartość średnią z wprowadzonych danych. Na wstępie omówimy klasyczny program, pokazany na kolejnym listingu. Program pobiera oceny i wylicza średnią. Nie jest znana z góry ilość danych. Wprowadzenie oceny równej 0 kończy program. W programie tablica **st[ ]** może przechowywać 50 ocen. Wprowadzanie danych do tablicy kontroluje pętla **do…while.** Po wpisaniu ostatniej oceny, zmienna **i** jest większa o jeden od liczby wprowadzonych ocen.

Listing 10.7 Wskaźniki, tablice, średnia

```
/* tablica - wartosc srednia */
#include <stdio.h>
#include <conio.h>
int main()
{int st[50],j, i = 0;
 int suma = 0;
 printf("\n 0 konczy wpisywanie ocen \n");
 do
   { printf("Wpisz ocene numer %d: ", i);
     scanf("%d", &st[i]);
   } while ( st[i++] > 0);
 j = i-1;
 for (i=0; i<j; i++)
  suma += st[i];
 printf(" srednia = %.1f", (float) suma/j);
 getche();
 return 0;
}
```

Sumowanie ocen wykonuje się w pętli **for**. Wydruk średniej oceny powoduje instrukcja:

```
printf(" srednia = %.1f", (float) suma/j);
```

W celu pokazania zastosowania zmiennych wskaźnikowych, program z listingu 10.7 będzie zmieniony tak, aby wykorzystać wskaźniki. Zmodyfikowany program do wyliczania średnich ocen pokazany jest na następnym listingu (10.8). Zmodyfikowane zostały wyrażenia odnoszące się do tablic. Instrukcja wczytywania danych i zapisywania ich w tablicy:

```
scanf("%d", &st[i]);
```

zamieniona została na instrukcję:

```
scanf("%d", st + i);
```

Warunek pętli **do…while**:

```
while ( st[i++] > 0);
```

został zmieniony na:

```
while ( *(st + i++) > 0);
```

Listing 10.8 Tablice, średnia, wskaźniki

```
/* srednia wartosc, wskazniki */
#include <stdio.h>
#include <conio.h>
#define ROZ 50

int main()
{int st[ROZ],j;
 int suma = 0;
 int i = 0;
 do
   {
    printf("Wpisz oceny %d: ", i);
    scanf("%d", st + i);
   } while ( *(st + i++) > 0);

 j = i-1;
 for (i=0; i<j; i++)
  suma += *(st + i);
 printf(" srednia = %.1f", (float) suma/j);
 getche();
 return 0;
}
```

Zmieniono także instrukcję sumowania:

```
suma += st[i];
```

na:

```
suma += *(st + i);
```

Jak łatwo zauważyć, wykorzystano wyrażenie **st + i** aby odwołać się do elementu tablicy (klasyczny sposób to **st[i]).** W tym wyrażeniu musimy zmieniać wartość **i** aby otrzymywać wartości kolejnych elementów tablicy. Nasuwa się pytanie czy nie można by stosować inkrementacji adresu tablicy, czyli użyć następującej konstrukcji:

```
while ( *(st++) > 0);   //błędne wyrażenie!!
```

W języku C nie można tego zastosować, ponieważ **st** jest stałą wskaźnikową. Jest to adres tablicy **st**, a wartość ta nie zmienia się w czasie działania programu. Próba zwiększenia tego adresu spowoduje wygenerowanie błędu kompilacji. Wiemy, że nie można inkrementować stałej, ale można inkrementować zmienne. Skorzystamy ze zmiennych wskaźnikowych, aby rozwiązać nasz problem.
Kolejny listing pokazuje jeszcze jedną modyfikację naszego programu.

Listing 10.9 Tablice, inkrementacja wskaźników

```
/*inkrementacja wskaznikow */
#include <stdio.h>
#include <conio.h>
#define ROZ 50
int main()
{int st[ROZ],j;
 int *wsk;
 int suma = 0;
 int i = 0;
 wsk = st;
 do
   {
    printf("Wpisz oceny %d: ", i++);
    scanf("%d", wsk);
   } while ( *(wsk++) > 0);

 wsk = st;
 j = i-1;
 for (i=0; i<j; i++)
       suma += *(wsk++);
 printf(" srednia = %.1f", (float) suma/j);
 getche();
 return 0;
}
```

Adres tablicy **st** został umieszczony w zmiennej wskaźnikowej o nazwie **wsk.**

Do tej zmiennej możemy odwoływać się jak do dowolnej innej zmiennej. Korzystamy ze zmiennej **wsk**, która jest adresem, zmienna ta wskazuje na element tablicy, a korzystając z operatora wyłuskania ***wsk** otrzymujemy zawartość tego adresu. Ponieważ **wsk** jest zmienną, możemy ją inkrementować. Wskaźnikowi **wsk** nadajemy wartość adresu tablicy **st** przed pętla for i ustawiamy go ponownie przed wejściem do pętli sumującej oceny. W obu pętlach (**for** i **do..while**) zwiększamy **wsk** za pomocą operatora inkrementacji ++, aby wskazywał na kolejny elementy tablicy **st**. Ponieważ **wsk** wskazuje na tablice typu **int**, operator inkrementacji ++ powoduje zwiększenie wskaźnika o dwa bajty.

## 10.10. Łańcuchy i wskaźniki

Łańcuchy są tablicami przechowującymi znaki, wobec tego istnieje ścisły związek pomiędzy łańcuchami i wskaźnikami.
W celu zilustrowania zastosowania wskaźników do operacji na łańcuchach zademonstrujemy program, który czyta napis z klawiatury a następnie poda adres, pod którym zapisana jest konkretna litera. Program korzysta z bibliotecznej funkcji **strchr(),** która umieszczona jest w pliku **string.h.**

Listing 10.10 Wskaźniki i łańcuchy, funkcja strchr()

```
/* napisy i wskazniki */
//podaje adres zapisu litery
#include <stdio.h>
#include <conio.h>
#include <string.h>

int main()
{
 char zn, napis[81];
 char *wsk;
 puts("Napisz zdanie: ");
 gets(napis);
 puts("Podaj litere: ");
 zn = getche();
 wsk = strchr(napis,zn);
 printf("\n Adres poczatkowy napisu %u" , napis);
 printf("\n Znak umieszczony jest pod adresem %u",
          wsk);
 printf("\n Znak jest na pozycji %d w napisie ",
          wsk-napis);
 getche();
 return 0;
}
```

Działanie tego programu może mieć następująca postać:

```
Napisz zdanie:
Ala ma kota
Podaj litere:
 k
Adres poczatkowy napisu 65408
Znak umieszczony jest pod adresem 65415
Znak jest na pozycji 7 w napisie
```

Funkcja **strchr()** zwraca wskaźnik do pierwszego wystąpienia wyspecyfikowanego znaku w napisie. Adres otrzymano dzięki instrukcji:

```
wsk = strchr(napis,zn);
```

Zmiennej wskaźnikowej **wsk** zostaje przypisany adres pierwszego wystąpienia znaku umieszczonego w zmiennej **zn,** w naszym przykładzie była to litera ‘k’. Funkcja biblioteczna **strchr()** wymaga dwóch parametrów: badanego łańcucha i szukanego znaku. Łańcuchy inicjalizuje się tak jak zwykłe tablice, korzystając z indeksów. Oczywiście łańcuchy możemy inicjalizować, jako wskaźniki. Program z kolejnego listingu (10.11) ilustruje to zagadnienie. Aby zainicjalizować łańcuch wykorzystano wyrażenie:

```
char *n1 = "Witaj ";
```

Listing 10.11 Wskaźniki i łańcuchy, funkcja strchr()

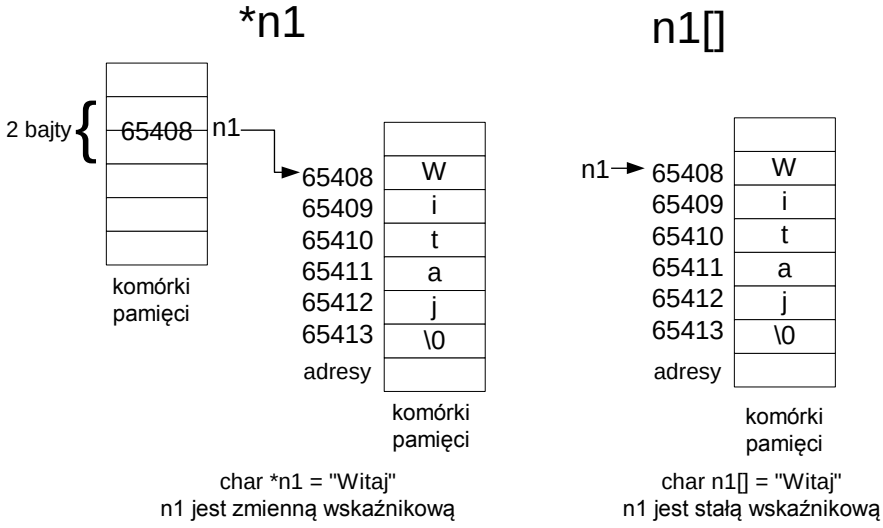
```
/* inicjalizowanie napisu - wskazniki */
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{char *n1 = "Witaj ";
 char n2[81], wynik[81];
 puts("napisz swoje imie: ");
 gets(n2);
 strcpy(wynik,n1);
 strcat(wynik,n2);
 printf("\n%s", wynik);
 getche();
 return 0;
}
```

Alternatywna postać inicjalizacji to:

```
char n1[] = ”Witaj”;
```

Miedzy tymi dwoma wyrażeniami występują subtelne różnice. W wersji tablicowej rezerwowana jest odpowiednia ilość komórek pamięci do przechowywania napisu. Adresowi pierwszego znaku z tablicy jest nadawana nazwa tablicy, w naszym przypadku **n1**.

Zgodnie z zasadami języka, **n1** jest stałą wskaźnikową, której adresu nie można zmieniać. W wersji wskaźnikowej rezerwowane są komórki pamięci w taki sam sposób oraz rezerwowane jest miejsce na zmienna wskaźnikową, jest tworzony wskaźnik, któremu nadawana jest w naszym przypadku nazwa **n1**. Na rysunku 10.4 pokazano schematycznie różnice w inicjalizacji łańcucha.




Rys.10.4 Dwa sposoby inicjalizowania łańcucha

Możliwość odwoływania się do łańcuchów poprzez ich adresy jest bardzo użyteczną własnością języka C. W kolejnym programie (listing 10.12) zademonstrujemy sortowanie imion wykorzystując sortowanie tablicy wskaźników do łańcuchów.
Po uruchomieniu, przykładowe działanie tego programu może być następujące:

```
podaj imie 1 : wacek
podaj imie 2 : ala
podaj imie 3 : ola
podaj imie 4 : [Enter]

Posortowane imiona :
ala
ola
wacek
```

Łańcuchy umieszczone są w tablicy dwuwymiarowej:

```
char imie[MAX1][MAX2];  //tablica napisow
```

Listing 10.12 Sortowanie łańcuchów

```
/* sortowanie napisow- wskazniki */
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX1 50    //maksymalna liczba imion
#define MAX2 20    //maksymalna dlugosc imienia
int main()
{char imie[MAX1][MAX2]; //tablica napisow
 char *wsk[MAX1];  //tablica wskaznikow
 char *wsk1;
 int ile, n1,n2;
 ile = 0;
 while (ile < MAX2)
    {printf("\n podaj imie %d :  ", ile+1);
     gets(imie[ile]);
     if (strlen(imie[ile]) == 0)   break;
     wsk[ile++] = imie[ile];
    }
for (n2 = 0; n2 < ile - 1; n2++)
   for (n1 = n2+1; n1 < ile; n1++)
      if (strcmp(wsk[n2],wsk[n1]) > 0)
      { wsk1 = wsk[n1];
        wsk[n1] = wsk[n2];
        wsk[n2] = wsk1;
      }
  printf("\n Posortowane imiona: \n\n");
  for (n2 = 0; n2 < ile; n2++)
     printf("%4d  %s \n", n2+1, wsk[n2]);
 getche();
 return 0;
}
```

W dyrektywie preprocesora ustalono, że w tym przypadku maksymalnie możemy sortować 50 imion, każde imię nie może składać się z więcej niż 20 znaków:

```
#define MAX1 50  //maksymalna liczba imion
#define MAX2 20  //maksymalna dlugosc imienia
```

Wczytywanie imion realizowane jest za pomocą pętli **while**:

```
  while (ile < MAX2)
    {
     printf("\n podaj imie %d :  ", ile+1);
     gets(imie[ile]);
     if (strlen(imie[ile]) == 0)   break;
     wsk[ile++] = imie[ile];
    }
```

Nie ustalamy ile imion będziemy sortować, warunkiem zakończenia pętli **while** jest wprowadzenie pustego łańcucha:

```
if (strlen(imie[ile]) == 0)   break;
```

Funkcja biblioteczna **strlen()** zwraca długość łańcucha. Jeżeli naciśniemy klawisz **Enter** bezpośrednio po żądaniu wpisania imienia, to długość łańcucha będzie równa zero. Funkcja **strlen()** zwróci tą zerową wartość i spowoduje wykonanie instrukcji **break**.
Podczas wczytywania imion, działanie instrukcji:

```
wsk[ile++] = imie[ile];
```

spowoduje, że adresy kolejnych łańcuchów, które są przechowywane w dwuwymiarowej tablicy **imie[ ][ ]** są przypisywane kolejnym elementom tablicy wskaźników **wsk[ ].** Mimo, że tablica **imie[ ][ ]** jest dwuwymiarowa, zapis **imie[]** nie jest błędem. Sortowanie realizowane jest w pętli for:

```
for (n2 = 0; n2 < ile - 1; n2++)
  for (n1 = n2+1; n1 < ile; n1++)
    if (strcmp(wsk[n2],wsk[n1]) > 0)
     {
      wsk1 = wsk[n1];
      wsk[n1] = wsk[n2];
       wsk[n2] = wsk1;
     }
```

Metoda sortowania nosi nazwę ***sortowania bąbelkowego***. Nie jest to najbardziej wydajny algorytm sortowania, ale jest prosty w implementacji. Sortowana jest tablica wskaźników **wsk[ ].** Tablica **imie[ ][ ]** pozostaje niezmieniona.
Należy zauważyć, że tablica **imie[ ][ ]** jest dwuwymiarowa, a my odwołujemy się do niej za pomocą jednego indeksu:

```
wsk[ile++] = imie[ile];
```

W języku C możemy traktować tablice dwuwymiarowe, jako tablice tablic. Wobec tego deklaracja:

```
char imie[MAX1][MAX2];  //tablica napisow
```

może być potraktowana, jako deklaracja jednowymiarowej tablicy o MAX1 elementach, z których każdy jest jednowymiarową tablica znaków o długości MAX2. Inaczej mówiąc **imie[0]** jest adresem pierwszego łańcucha w tablicy, **imie[1]** jest adresem drugiego łańcucha, itd.

## 10.11. Wskaźniki do wskaźników

Język C ma możliwość tworzenia wskaźników wskazujących na inne wskaźniki. Dzięki temu mamy możliwość tworzenia skomplikowanych struktur danych. Wskaźniki do wskaźników zastosujemy w kolejnym programie. Zadaniem naszym jest napisanie programu, dzięki któremu będziemy mogli do

każdego elementu tablicy dwuwymiarowej dodać stały czynnik. Najpierw opracujemy program w wersji tablicowej ( posługiwać się będziemy indeksami).

Listing 10.13 Operacje tablicowe, wskaźniki do wskaźników

```
// Tablica dwuwymiarowa, wersja tablicowa
#include <stdio.h>
#include <conio.h>
#define WIER 3
#define KOL 4
int main()
{int tab[WIER][KOL] =
     { {  14,  17,  27,  22},
       {  24,  28,  37,  33},
       {  34,  39,  47,  44} };
 int nj,nk;
 for (nj=0; nj<WIER; nj++)    //dodaje 100 elementu
   for (nk=0; nk<KOL; nk++)
      tab[nj][nk] = tab[nj][nk] + 100;
 printf("\n");
 for (nj=0; nj<WIER; nj++)     //drukuje nowa tablice
    { for (nk=0; nk<KOL; nk++)
        printf("%d ",tab[nj][nk]);
      printf("\n");
    }
getche();
return 0;
}
```

Deklaracja i inicjalizacja tablicy o rozmiarze 3x4 ma postać:

```
int tab[WIER][KOL] =
     {   {  14,  17,  27,  22},
         {  24,  28,  37,  33},
         {  34,  39,  47,  44} };
```

Do każdego elementu tej tablicy dodawana jest liczba 100:

```
for (nj=0; nj<WIER; nj++)     //dodaje 100
  for (nk=0; nk<KOL; nk++)
     tab[nj][nk] = tab[nj][nk] + 100;
```

Wydruk tablicy realizowany jest w pętli **for**:

```
for (nj=0; nj<WIER; nj++)     //drukuje nowa tablice
   { for (nk=0; nk<KOL; nk++)
          printf("%d ",tab[nj][nk]);
     printf("\n");
   }
```

Możemy opracować program wykonujący to samo zadanie, ale korzystający ze wskaźników.

Zasadniczym zadaniem jest wykorzystanie wskaźnika w celu odwołania się do konkretnego elementu tablicy, np. chcemy odwołać się do elementu **tab[2][2],** który w naszym przykładzie ma wartość 47. Nasza tablica jest typy **int**, czyli jeden element wymaga 2 bajtów pamięci. Wiemy, że adresem całej tablicy jest **tab**. Jeżeli przyjmiemy, że nasza tablica zaczyna się od adresu 64100 w pamięci to mamy:

```
tab == 64100
```

W każdym wierszu mamy 4 elementy, do zapisania wiersza potrzebujemy 8 bajtów. Oznacza to, że po każdych 8 bajtach zaczyna się nowy wiersz. Wiemy również, że kompilator wykorzystuje arytmetykę wskaźnikową. Potrafi interpretować takie wyrażenia jak np.:

```
tab + 1
```

Dla tego przykładu, kompilator pobiera adres początku tablicy (w naszym komputerze było to 64100) i dodaje liczbę bajtów w wierszu ( 4 kolumny razy 2 bajty, czyli 8 bajtów). W ten sposób wiemy, że **tab + 1** jest interpretowane jako adres 64108. Należy jeszcze rozważyć, w jaki sposób odwołać się do pojedynczego elementu w wierszu. Wiemy, że adresem tablicy tworzonej przez np. trzeci wiersz tablicy **tab[][]** jest **tab[2]** albo inaczej **tab + 2** w notacji wskaźnikowej. Adresem pierwszego elementu tej tablicy jest **&tab[2][0],** czyli ***(tab + 2).** Oba wyrażenia wskaźnikowe **tab + 2** i ***(tab +2)** odnoszą się do zawartości tego samego adresu.
Wiemy wiec, że

*do elementu tablicy dwuwymiarowej można się odwoływać*
*poprzez wskaźnik do wskaźnika*

Relacja pomiędzy zapisem indeksowym i wskaźnikowym jest następująca:

```
tab[i][j] == *(*(tab+i) + j)
```

Program wykonujący takie samo zadanie jak poprzedni program może mieć postać pokazaną na kolejnym listingu (10.14).
Dodawanie wartości 100 do każdego elementu realizowane jest w następującym bloku:

```
for (nj=0; nj<WIER; nj++)         //dodaje 100
   for (nk=0; nk<KOL; nk++)
      *(*(tab+nj)+nk) = *(*(tab+nj)+nk) + 100;
```

Wydruk nowych elementów realizuje blok instrukcji:

```
for (nj=0; nj<WIER; nj++)      //drukuje nowa tablice
    { for (nk=0; nk<KOL; nk++)
        printf("%d ", *(*(tab +nj) + nk) );
      printf("\n");
    }
```

Listing 10.14 Operacje tablicowe, wskaźniki do wskaźników

```
// Tablica dwuwymiarowa, wersja wskaznikowa
#include <stdio.h>
#include <conio.h>
#define WIER 3
#define KOL 4

int main()
{int tab[WIER][KOL] =
     {  {  11,  12,  13,  14},
        {  24,  28,  37,  33},
        {  34,  39,  47,  44} };
 int nj,nk;
 for (nj=0; nj<WIER; nj++)    //dodaje 100
    for (nk=0; nk<KOL; nk++)
      *(*(tab+nj)+nk) = *(*(tab+nj)+nk) + 100;
 printf("\n");
 for (nj=0; nj<WIER; nj++)     //drukuje nowa tablice
    { for (nk=0; nk<KOL; nk++)
        printf("%d ", *(*(tab +nj) + nk) );
      printf("\n");
    }
getche();
return 0;
}
```

# ROZDZIAŁ 11

## STRUKTURY

## 11.1. Wstęp

**Struktura** jest obiektem złożonym z kilku zmiennych, najczęściej różnego typu, często dla wygody zgrupowanych pod jedna nazwą. Zmienne strukturalne pozwalają na lepsze zorganizowanie skomplikowanych danych, dzięki czemu można łatwiej je przetwarzać.

## 11.2. Wprowadzenie do struktur

Podczas projektowania programu bardzo często musimy decydować, w jaki sposób będą reprezentowane dane. Proste dane czy tablice w wielu przypadkach nie są optymalnym wyborem. Język C oferuje tzw. ***zmienne strukturalne*** (ang. *structure variables*). ***Struktury*** w języku C są na tyle uniwersalne, że pozwalają na tworzenie całkiem skomplikowanych struktur danych. Struktury najczęściej wykorzystywane są do definicji rekordów danych przechowywanych w plikach. Wskaźniki i struktury ułatwiają tworzenie bardziej złożonych struktur danych jak listy powiązane, kolejki, stosy i drzewa.

## 11.3. Deklaracja typedef

W języku C mamy podstawowe typy danych (np**. int, float**, itp.) a także pochodne typy danych (np. tablice czy wskaźniki). Pewnym udogodnieniem w języku C jest możliwość definiowania własnych typów danych. Takim mechanizmem jest deklaracja **typedef,** dzięki której deklarujemy własne nazwy dla podstawowych lub pochodnych typów danych.
Deklaracja **typedef** ma postać:

```
typedef  std_nazwa_typu  własna_nazwa_typu
```

Powody stosowania mechanizmu **typedef** mogą być różne. Możemy np. długie deklaracje zastąpić krótką deklaracją:

```
typedef unsigned long int  uli;
```

Stworzony został synonim, wystąpienie napisu **uli** powoduje, że w deklaracji:

```
uli  liczba1, liczba2;
```

zmienne **liczba1** i **liczba2** są zadeklarowane, jako zmienne typu **unsigneg long int**. Nazwy zastępcze służą często do ukrywania informacji a także pozwalają na pisanie bardziej czytelnych programów.

Krótki program (listing 11.1) demonstruje użycie mechanizmu **typedef**. Program pozwala na wykonywanie operacji na wektorach. W programie ograniczono się do obliczenia iloczynu skalarnego dwóch wektorów ( **a** i **b**) o trzech składowych oraz do wykonania operacji dodawania dwóch wektorów (**a** i **b**) w wyniku, czego otrzymujemy trzeci wektor ( **c** ). Wymiar wektora ustala dyrektywa preprocesora:

```
#define N 3
```

Listing 11.1. Tworzenie typu specjalnego, użycie konstrukcji typedef

```
#include <stdio.h>
#include <conio.h>
#define N 3
  typedef int skalar;
  typedef skalar wektor[N];
  skalar il_skalar(wektor, wektor);
  void suma_tab(wektor a, wektor b);
int main()
{ wektor a,b,c;
  skalar wynik ;
  int i;
  printf ("\npodaj skladowe wektora a :");
    for (i=0; i<N; i++)   {
        printf("\na[%2d]= ",i);
        scanf("%d",&a[i]);
     }
  printf ("\npodaj skladowe wektora b :");
     for (i=0; i<N; i++)   {
         printf("\nb[%2d]= ",i);
         scanf("%d",&b[i]);
     }
  wynik = il_skalar(a,b);
  printf("\n iloczyn skalarny = %d\n",wynik);
  suma_tab(a,b);
  getche();
  return 0;
}
skalar il_skalar(wektor a, wektor b)
{ int i;
  skalar iloczyn = 0;
   for (i = 0; i <N; i++)      iloczyn += a[i] *b[i];
  return(iloczyn);
}
void suma_tab(wektor a, wektor b)    // c = a + b
{ int i ;
  wektor c;
    printf("\nsuma elementow c = a + b \n");
      for (i = 0; i <N; i++)    {
         c[i] = a[i] + b[i];
         printf("\n c[%d] =  %d",i,c[i]);
      }
}
```

Za pomocą słowa kluczowego **typedef** został utworzony nowy typ - **skalar**, za pomocą tego typu utworzono ponownie nowy, następny typ - **wektor**:

```
typedef int skalar;
typedef skalar wektor[N];
```

Wprowadzono dwie funkcje - do obliczenia iloczynu skalarnego i dodania wektorów. Prototypy tych funkcji mają postać:

```
skalar il_skalar(wektor, wektor);
void suma_tab(wektor a, wektor b);
```

W funkcji **main()** wprowadzamy wartości składowych dwóch wektorów, a następnie są wywoływane funkcje:

```
wynik = il_skalar(a,b);
suma_tab(a,b);
```

Wynikiem działania tego programu jest przykładowy wydruk:

```
podaj skladowe wektora a :
a[0]= 1
a[1]= 1
a[2]= 1
podaj skladowe wektora b :
b[0]= 2
b[1]= 2
b[2]= 2
iloczyn skalarny = 6
suma elementow c = a + b
c[0]= 3
c[1]= 3
c[2]= 3
```

Zastosowanie substytucji **typedef** do zdefiniowania typu **skalar** i **wektor**, pozwala programiście koncentrować się na aplikacji raczej niż na kodowaniu w języku C. Gdy zajdzie potrzeba zmiany typu (np. elementy będą typu **float** lub **double**), wystarczy zmienić tylko jedną linię kodu:

```
typedef int skalar;
```

na nową postać:

```
typedef float skalar;
```

lub

```
typedef double skalar;
```

Oczywiście należy także pamiętać o obsłudze funkcji wczytujących dane!

## 11.4. Deklaracja struktury

Jak pamiętamy, tablice pozwalają na przechowywanie wielu danych tego samego typu. Język C pozwala na zgrupowanie danych różnego typu - istnieje specjalny typ danych zwanych ***strukturą***. Struktura składa się z wielu elementów danych, które mogą być różnego typu zgrupowanych razem.

Zapis deklaracji struktury jest następujący:

```
struct nazwa
  {  typ_danych nazwa1;
     typ_danych nazwa2;
     ..............................
     typ_danych nazwaN;
   };      //UWAGA na średnik
```

Przykładowa struktura może mieć następującą postać:

```
struct pacjent         //definicja struktury "pacjent"
  {int   nr_badania;    //elementy struktury
   char  *imie ;        //elementy struktury
   float koszt;         //elementy struktury
  } ;
```

W ten sposób została utworzona (za pomocą słowa kluczowego **struct**) struktura danych o nazwie **pacjent**. Struktura ta składa się z następujących elementów:

```
nr_badania (typu int)
imie (typu char)
koszt (typu float)
```

Definicja struktury na początku programu wprowadza nowy typ o nazwie **pacjent.** Jeżeli chcemy wykorzystać definicje struktury do nowych zmiennych (np. o nazwach **pt1, pt2**) to deklarujemy to tak jak dla typowych zmiennych:

```
   struct pacjent pt1,pt2;
```

Ta instrukcja rezerwuje miejsce w pamięci komputera - kompilator ustali ilość bitów potrzebnych do pomieszczenia wszystkich elementów struktury.
W języku C możemy połączyć deklarację i definicję nowych zmiennych typu strukturalnego w jedną instrukcję:

```
struct pacjent         //definicja struktury "pacjent"
  {int   nr_badania;    //elementy struktury
   char  *imie ;        //elementy struktury
   float koszt;         //elementy struktury
  } pt1,pt2;     //deklaracje zmiennychstrukturalnych
```

Istnieje jeszcze jeden sposób deklaracji - za pomocą słowa kluczowego **typedef**:

```
typedef struct
{  int   nr_badania;    //elementy struktury
   char  *imie ;        //elementy struktury
   float koszt;         //elementy struktury
} pacjent;
```

Tego typu deklaracja umożliwia tworzenie nowych nazw dla typów danych. Deklaracja zmiennych może mieć postać:

```
     pacjent pt1,pt2;
```

Dostęp do składowych (pól, elementów) struktury można uzyskać dwoma sposobami:

- za pomocą operatora kropki ( . )
- za pomocą operatora minus, znak większości ( –> )

Operator kropki umożliwia łatwy dostęp do elementów struktury. W przykładowym programie zastosowano ten operator do przypisania wartości elementom struktury. W naszym programie najpierw utworzona jest struktura o nazwie **pacjent**. Zawiera ona trzy elementy:

```
int   nr_badania;       //numer badania
char  *imie ;           //imię pacjenta
float koszt;            //koszt badania
```

Następnie musimy wprowadzić dane, innymi słowy nadać wartość poszczególnym elementom struktury. Pokazane jest to w przykładowym programie.

Listing 11.2. Struktury, dostęp do składowych

```
#include <stdio.h>
#include <conio.h>
int main()
{ struct pacjent        //definicja struktury "pacjent"
  {int   nr_badania;    //elementy struktury
   char  *imie ;
   float koszt;
  } ;
  struct pacjent pt1;             //deklaracja
  pt1.nr_badania = 1111;          //nadanie wartosci
  pt1.imie = "waclaw";
  pt1.koszt = 35.40;
  printf("\n       imie   %s",pt1.imie);
  printf("\n nr badania   %d",pt1.nr_badania);
  printf("\n      koszt   %5.2f zl",pt1.koszt);
  getche();
  return 0;
}
```

Po uruchomieniu tego programu mamy wydruk:

```
imie   waclaw
nr badania  1111
koszt  35.40 zl
```

Przypisanie wartości elementom struktury **pacjent** następuje w instrukcjach:

```
pt1.nr_badania = 1111;  //nadanie wartosci
pt1.imie = "waclaw";
pt1.koszt = 35.40;
```

W języku C może być dowolna ilość zmiennych wbudowanego typu strukturalnego. W programie z listingu 11.3 zdefiniowana jest rozbudowana struktura, mamy także dwie zmienne: **pt1** i **pt2**, które są zmiennymi typu **struct pacjent**.

Listing 11.3. Rozbudowane struktury

```
#include <stdio.h>
int main()
{
 struct pacjent   //definicja struktury "pacjent"
     { int   nr_badania;
       char  *nazwisko ;
       char  *imie ;
       int   rok_urodzenia;
       char  *typ_badania;
       float koszt;
     } pt1,pt2;

  pt1.nr_badania = 1111;          //nadanie wartosci
  pt1.nazwisko = "Kowalski";
  pt1.imie = "Waclaw";
  pt1.rok_urodzenia = 1944;
  pt1.typ_badania = "CT";
  pt1.koszt = 35.40;
  pt2.nr_badania = 1112; //nadanie wartosci
  pt2.nazwisko = "Zalewska";
  pt2.imie = "Marianna";
  pt2.rok_urodzenia = 1974;
  pt2.typ_badania = "USG";
  pt2.koszt = 25.50;

  printf("\n------------------------------");
  printf("\n  %s %s %d r.",pt1.nazwisko,
            pt1.imie,pt1.rok_urodzenia);
  printf("\n  nr badania %d , typ  %s",pt1.nr_badania,
            pt1.typ_badania);
  printf("\n  koszt   %5.2f zl",pt1.koszt);
  printf("\n------------------------------");
  printf("\n  %s %s %d r.",pt2.nazwisko,
            pt2.imie,pt2.rok_urodzenia);
  printf("\n  nr badania %d , typ  %s",pt2.nr_badania,
            pt2.typ_badania);
  printf("\n  koszt   %5.2f zl",pt2.koszt);
}
```

W tym programie połączono deklarację typu strukturalnego i definicję zmiennych strukturalnych w jedną instrukcję:

```
struct pacjent   //definicja struktury "pacjent"
  {int   nr_badania;
   char  *nazwisko ;
   char  *imie ;
   int   rok_urodzenia;
   char  *typ_badania;
   float koszt;
  } pt1,pt2;
```

Odwołanie do elementów struktury ma postać:

```
pt1.nazwisko = "Kowalski";
```

Po uruchomieniu programu otrzymujemy następujący wydruk:

```
- - - - - - - - - - - - - - - - - - -
Kowalski Waclaw 1944 r.
nr badania 1111 , typ CT
koszt  35.40 zl
- - - - - - - - - - - - - - - - - - -
Zalewska Marianna 1974 r.
nr badania 1112 , typ USG
koszt  25.50 zl
```

W następnym programie (listing 11.4) pokazano program, gdzie dane wprowadzane są z klawiatury. Należy zwrócić uwagę na fakt użycia jedynie funkcji **gets()** do wczytywania danych z klawiatury.

Listing 11.4. Struktury, obsługa wejścia

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main()
{
 struct pacjent   //definicja struktury "pacjent"
  {
   int   nr_badania;
   char  *nazwisko ;
   char  *imie ;
   int   rok_urodzenia;
   char  *typ_badania;
   float koszt;
  } pt1,pt2;
  char numstr [81];
  printf("\npacjent nr 1");
  printf("\npodaj nr badania : ");
  gets(numstr);
  pt1.nr_badania = atoi(numstr);
```

```
    printf("\npodaj nazwisko : ");
    gets(pt1.nazwisko);
    printf("\npodaj imie : ");
    gets(pt1.imie);
    printf("\npodaj rok urodzenia : ");
    gets(numstr);
    pt1.rok_urodzenia = atoi(numstr);
    printf("\npodaj typ badania : ");
    gets(pt1.typ_badania);
    printf("\npodaj koszt badania : ");
    gets (numstr);
    pt1.koszt = atof(numstr);
    printf("\npacjent nr 2");
    printf("\npodaj nr badania : ");
    gets(numstr);
    pt2.nr_badania = atoi(numstr);
    printf("\npodaj nazwisko : ");
    gets(pt2.nazwisko);
    printf("\npodaj imie : ");
    gets(pt2.imie);
    printf("\npodaj rok urodzenia : ");
    gets(numstr);
    pt2.rok_urodzenia = atoi(numstr);
    printf("\npodaj typ badania : ");
    gets(pt2.typ_badania);
    printf("\npodaj koszt badania : ");
    gets (numstr);
    pt2.koszt = atof(numstr);
    printf("\n------------------------------");
    printf("\n Nazwisko i imie : %s %s \n",
             pt1.nazwisko,pt1.imie);
    printf(" rok urodzenia   : %6d   \n ",
             pt1.rok_urodzenia);
    printf("nr badania      : %6d\n",pt1.nr_badania);
    printf(" typ badania  :   %s\n", pt1.typ_badania);
    printf(" koszt        :   %5.2f zl\n",pt1.koszt);
    printf("\n------------------------------");
    printf("\n Nazwisko i imie : %s %s \n",
              pt2.nazwisko,pt2.imie);
    printf(" rok urodzenia   : %6d   \n ",
              pt2.rok_urodzenia);
    printf("nr badania      : %6d\n",pt2.nr_badania);
    printf(" typ badania    :   %s\n",pt2.typ_badania);
    printf(" koszt          :   %5.2f zl\n",pt2.koszt);
    getche();
    return 0;
}
```

Jednolite zastosowanie funkcji **gets()** zapobiega potencjalnym problemom z wczytywaniem danych, jakie mogłyby pojawić się w przypadku stosowania funkcje **gets()** jednocześnie z funkcją **scanf()**. Wykorzystano dwie funkcje biblioteczne:

```
printf("\npodaj nr badania : ");
gets(numstr);
pt1.nr_badania = atoi(numstr); // liczby całkowite
```

oraz

```
printf("\npodaj koszt badania : ");
gets (numstr);
pt1.koszt = atof(numstr);   // liczby rzeczywiste
```

## 11.5. Tablice struktur

Już na pierwszy rzut oka widać, że pokazany na poprzednim listingu program jest rozwlekle napisany. Znaczne fragmenty kodu są powtórzone. Jak można się spodziewać, wyjściem jest zastosowanie tablicy struktur. Użycie tablicy struktur pokazane jest w następnym programie. Struktura **pacjent** odnosi się do jednego pacjenta, ale możemy stworzyć tablicę zawierającą wielu pacjentów. Program jest nieco bardziej skomplikowany, można dodawać dane pacjentów a także drukować pełna listę danych.

Listing 11.5. Tablice struktur

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define TRUE 1
void nowy_pacjent(void);
void drukuj_liste(void);

 struct pacjent
    {int   nr_badania;
     char  nazwisko[30] ;
     char  imie[15] ;
     int   rok_urodzenia;
     char  typ_badania[10];
     float koszt;
    } ;
struct pacjent lpc[25];   // tablica 25 struktur
int n = 0;

int main()
{ char znak;
  while (TRUE)
   {printf("\n wybierz 'n' aby wprowadzic nowe
            nazwisko");
```

```
    printf("\n wybierz 'l' aby wydrukowac liste
             pacjentow");
    printf("\n wybierz 'q' aby zakonczyc program");
    printf("\n wybor : ");
    znak = getche();
    switch (znak)
      {case 'n' :
            nowy_pacjent(); break;
       case 'l' :
            drukuj_liste(); break;
       case 'q' :
            exit(0);
       default :
            puts("\n wprowadz tylko n,l lub q");
      }  // koniec instrukcji switch
   }  // koniec petli while
  return 0;
} // koniec funkcji main()

void nowy_pacjent(void)
 {char numstr [81];
  printf("\npacjent nr %d ", n+1);
  printf("\npodaj nr badania : ");
  gets(numstr);
  lpc[n].nr_badania = atoi(numstr);
  printf("\npodaj nazwisko : ");
  gets(lpc[n].nazwisko);
  printf("\npodaj imie : ");
  gets(lpc[n].imie);
  printf("\npodaj rok urodzenia : ");
  gets(numstr);
  lpc[n].rok_urodzenia = atoi(numstr);
  printf("\npodaj typ badania : ");
  gets(lpc[n].typ_badania);
  printf("\npodaj koszt badania : ");
  gets (numstr);
  lpc[n].koszt = atof(numstr);
  n++;
} //koniec funkcji nowy_pacjent()

void drukuj_liste(void)
{int j;
 if (n < 1)  printf("\n pusta lista");
 for (j=0; j < n; j++)
  {printf("\n------------------------------");
   printf("\n Pacjent Nr %d",j+1);
   printf("\n Nazwisko i imie : %s %s \n",
           lpc[j].nazwisko,lpc[j].imie);
   printf(" rok urodzenia   : %6d   \n ",
```

```
            lpc[j].rok_urodzenia);
    printf("nr badania   : %6d\n",lpc[j].nr_badania);
    printf(" typ badania :   %s\n",lpc[j].typ_badania);
    printf(" koszt       :   %5.2f zl\n",lpc[j].koszt);
   } //koniec petli for
}   // koniec funkcji drukuj_liste()
```

W naszym programie struktura ma postać:

```
struct pacjent
  {
   int   nr_badania;
   char  nazwisko[30] ;
   char  imie[15] ;
   int   rok_urodzenia;
   char  typ_badania[10];
   float koszt;
  } ;
```

Tablica struktur zdefiniowana jest następująco:

```
struct pacjent lpc[25];   // tablica 25 struktur
```

Instrukcja rezerwuje w pamięci miejsce na 25 struktur typu **pacjent.** Struktura zdefiniowana jest, jako globalna. Dzięki temu, funkcje:

```
void nowy_pacjent(void);
void drukuj_liste(void);
```

mają dostęp do tych struktur. Dostęp do elementów struktury jest prosty:

```
lpc[n].nr_badania = atoi(numstr);
```

Odwołujemy się do tablicy **lpc[ ]** za pomocą indeksu, używamy operatora kropki, po którym umieszczona jest nazwa elementu struktury.

# ROZDZIAŁ 12

## BITY, POLA BITOWE, UNIE

## 12.1. Wstęp

Język C dostarcza wiele narzędzi, dzięki którym można manipulować poszczególnymi bitami w wartościach przechowywanych w pamięci komputera. W języku C występują **operatory bitowe**, dzięki którym można manipulować bitami. Można także manipulować grupami bitów dzięki polom bitowym, które tworzone są za pomocą struktury. Struktury w języku C są na tyle uniwersalne, że pozwalają na tworzenie całkiem skomplikowanych struktur danych. Struktury najczęściej wykorzystywane są do definicji rekordów danych przechowywanych w plikach. Wskaźniki i struktury ułatwiają tworzenie bardziej złożonych struktur danych jak listy, powiązane, kolejki, stosy i drzewa. ***Unia*** (ang. *union*) jest typem, który pozwala przechowywać różne rodzaje danych w tym samym obszarze pamięci (jednak nie równocześnie). Dzięki uniom możliwe jest na przykład utworzenie tablicy jednostek o jednakowej długości, z których każda może przechowywać dane innego typu.

## 12.2. Reprezentacja binarnych liczb całkowitych

W arytmetyce komputerowej liczby wyrażane są w systemie dwójkowym. Do zapisu liczby binarnej stosujemy dwa znaki – 0 i 1, te symbole nazywane są także bitami. Za pomocą zer i jedynek możemy zapisać każdą liczbę, w systemach komputerowych mamy duże ograniczenia – bity są zorganizowane w grupy, 8 bitów tworzy jeden bajt. Liczby całkowite zapisujemy najczęściej za pomocą 2 lub 4 bajtów, co powoduje, że maksymalna liczba całkowita nie jest zbyt duża.

Osiem bitów tworzących jeden bajt numeruje się liczbami od 0 do 7. Bit numer 7 nazywamy bitem najbardziej znaczącym. Każdy z bitów odpowiada określonej potędze liczby 2. Jeżeli wszystkie bity są jedynkami to otrzymamy liczbę binarną 11111111, co w systemie dziesiętnym odpowiada liczbie 255:

$$1x2^7 + 1x2^6 + 1x2^5 + 1x2^4 + 1x2^3 + 1x2^2 + 1x2^1 + 1x2^0 =$$
$$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

Na rysunku 12.1 przedstawione są numery bitów i ich wartości.

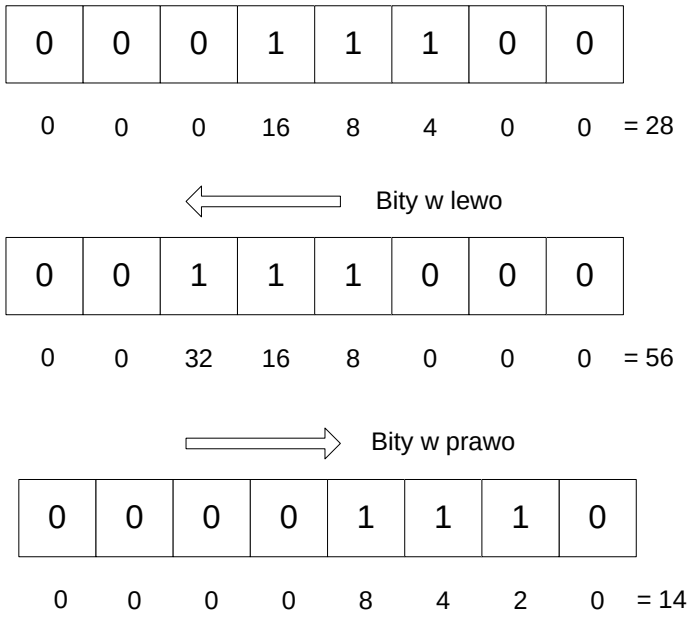
| Numer bitu | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| bajt | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| Potęga | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| i wartość bitu | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

Rys.12.1 Kodowanie liczby jednobajtowej.

W przykładzie pokazanym na rysunku 12.1 włączone są bity nr 3, 2 i 0. Wartość dziesiętna tego bajta wynosi 8 + 4 + 1 = 13.

Najmniejszą liczbą, jaką można zapisać w jednym bajcie jest 00000000, czyli zero. W jednym bajcie możemy zapisać liczby z przedziału od 0 do 255. Reprezentacja liczb ze znakiem jest określona sprzętowo. Najprostszym sposobem zapisania znaku liczby w bajcie jest wydzielenie jednego bitu. Zwykle jest to bit najbardziej znaczący. W takiej sytuacji do zapisu liczby pozostaje 7 bitów, czyli możemy zapisywać liczby z przedziału -127 do +127. Taki sposób zapisywania liczb ze znakiem nazywany jest *reprezentacją znak-moduł*, 00000001 to 1, a 100000001 to -1.

Przesunięcie bitów w bajcie udostępnia szybki sposób mnożenia i dzielenia przez potęgi dwójki. W języku C są odpowiednie narzędzia do przesuwania bitów. Operatory przesunięcia (ang. *shift operatorors*) pozwalają przesuwać bity w lewo lub w prawo. Operator przesunięcia w lewo ma postać << (dwa znaki mniejszości) , a operator przesunięcia w prawo jest reprezentowany przez >> (dwa znaki większości).




Rys.12.2 Operacja przesuniecie bitów o jedno miejsce

Na rysunku 12.2 zademonstrowano efekt przesunięcia bitów reprezentujących liczbę 28 o jedno miejsce w lewo, w wyniku, czego otrzymano liczbę 56. Przesunięcie bitów liczby 28 o jedno miejsce w prawo jest równoważne podzieleniu tej liczby przez 2.

## 12.3. Operatory bitowe

W języku C mamy operatory, zwane **operatorami bitowymi**, dzięki którym możemy traktować zmienną, jako zbiór bitów i wykonywać operacje na poszczególnych bitach. Zmienna może być typu całkowitego lub znakowego. Nie wolno używać operatorów bitowych do zmiennych typu **float** (**double**).
Lista operatorów bitowych jest następująca:

| Operator | symbol | opis |
|---|---|---|
| operatory logiczne: | & | bitowa koniunkcja (AND) |
| | \| | bitowa alternatywa (OR) |
| | ^ | bitowa różnica symetryczna (XOR) |
| | ~ | uzupełnienie jedynkowe |
| operatory przesunięcia | << | przesunięcie w lewo |
| | >> | przesunięcie w prawo |

Należy pamiętać, że operacje bitowe w ogólności są maszynowo zależne. Do naszych rozważań przyjmiemy, że bajt składa się z 8 bitów. Słowo składa się z 2 bajtów (16 bitów) lub 4 bajtów (32 bity).
Operatory bitowe działają na dwóch elementach tego samego typu (operator uzupełnienia ~ jest operatorem unarnym).

Jak wiemy zazwyczaj liczba całkowita potrzebuje 2 bajtów (bity numerowane są od 0 do 15). Operator bitowy działając na tych liczbach, porównuje odpowiadające sobie bity, i w zależności od charakteru daje odpowiedni wynik – powstaje nowy zapis bitowy.

## 12.4. Systemy liczbowe

W obliczeniach używamy systemu dziesiętnego, w obliczeniach komputerowych mamy do dyspozycji systemy binarne (podstawa 2), ósemkowe (podstawa 8) oraz szesnastkowe (podstawa 16). Operatory bitowe operują na liczbach przedstawianych w systemie binarnym – liczba taka składa się z zer i jedynek. Operowanie liczbami binarnymi jest dość kłopotliwe, stąd pomysł operowania na liczbach ósemkowych lub szesnastkowych. Jest wiele sposobów konwersji liczby z jednego systemu na drugi. Wygodnym sposobem przechodzenia z systemu binarnego na ósemkowy lub szesnastkowy jest grupowanie bitów. Należy mieć tylko odpowiednie tablice. Jeżeli mamy liczbę dziesiętną np. 13 to jej zapis w systemie binarnym może mieć postać:

$$13_{dec} = 1101_{bin}$$

Aby z liczby binarnej otrzymać liczbę ósemkową możemy zgrupować bity w trójki (brakujące cyfry uzupełniamy zerami):

$$1101 = 001\ 101$$

| Liczba dziesiętna | kombinacja bitów | liczba ósemkowa |
|---|---|---|
| 0 | 000 | 0 |
| 1 | 001 | 1 |
| 2 | 010 | 2 |
| 3 | 011 | 3 |
| 4 | 100 | 4 |
| 5 | 101 | 5 |
| 6 | 110 | 6 |
| 7 | 111 | 7 |

W tabeli znajdujemy, że grupie binarnej 001 odpowiada cyfra ósemkowa 1, a grupie 101 odpowiada cyfra ósemkowa 5. Zatem:

$$1101_{bin} = 15_{oct}$$

Podobnie wykonujemy konwersję do liczb szesnastkowych – tworzymy grupy czwórkowe.

| Liczba dziesiętna | kombinacja bitów | liczba szesnastkowa |
|---|---|---|
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |
| 10 | 1010 | A |
| 11 | 1011 | B |
| 12 | 1100 | C |
| 13 | 1101 | D |
| 14 | 1110 | E |
| 15 | 1111 | F |

Liczba dziesiętna np. 133 ma reprezentacje binarną:

$$133_{dec} = 10000101_{bin}$$

Konwersji do liczby szesnastkowej możemy dokonać grupując bity w czwórki:

1000 0101

grupie 1000 odpowiada cyfra szesnastkowa 8, a grupie 0101 odpowiada cyfra 5, mamy, zatem

$$10000101_{bin} = 85_{hex}$$

Jeżeli mamy liczbę szesnastkową np. A9 to liczbę w zapisie dziesiętnym możemy otrzymać np. w następujący sposób.
Korzystając z tabeli mamy:

$$A9_{hex} = 1010\ 1001_{bin}$$

Tworzymy tabelkę:

| Nr bitu | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Zapis binarny | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| Konwersja | $1*2^7$ | $0*2^6$ | $1*2^5$ | $0*2^4$ | $1*2^3$ | $0*2^2$ | $0*2^1$ | $1*2^0$ |
| Sumy | 128 | 0 | 32 | 0 | 8 | 0 | 0 | 1 |

Po zsumowaniu sum cząstkowych otrzymujemy liczbę dziesiętną:

$$128 + 32 + 8 + 1 = 169$$

Mamy, zatem:

$$A9_{hex} = 169_{dec}$$

Zamiana liczby z systemu dziesiętnego na binarny wymaga innego algorytmu. Wykorzystując funkcję rekurencyjną możemy mieć efektywny program przedstawiający w systemie dwójkowym (binarnym) dziesiętne liczby całkowite. Tego typu program pokazany jest na listingu 12.1.
Po uruchomieniu programu możemy mieć następujący wynik:

```
Wyświetlanie liczb binarnie, q konczy program
Podaj liczbe: 0
 postac binarna liczby 16 = 10000
Wyświetlanie liczb binarnie, q konczy program
Podaj liczbe: 255
 postac binarna liczby 255 = 11111111
Wyświetlanie liczb binarnie, q konczy program
Podaj liczbe: 1024
 postac binarna liczby 1024 = 10000000000
Wyświetlanie liczb binarnie, q konczy program
Podaj liczbe: q
```

Listing 12.1. Konwersja liczb dziesiętnych na binarne, rekurencja

```
#include <stdio.h>
#include <conio.h>
void bin(int);

int main()
{
  int liczba;
  printf("\n Wyswietlanie liczb binarnie, q konczy
          program");
  printf("\nPodaj liczbe: ");
  while (scanf("%d", &liczba) ==1)
    {printf(" postac binarna liczby %d = ",liczba);
     bin(liczba);
     printf("\nPodaj liczbe: ");
    }
  getche();
  return 0;
}

void bin(int x)
 { int b;
   b = x % 2;
   if (x>=2)   bin(x / 2);
  putchar('0' + b);
  return;
 }
```

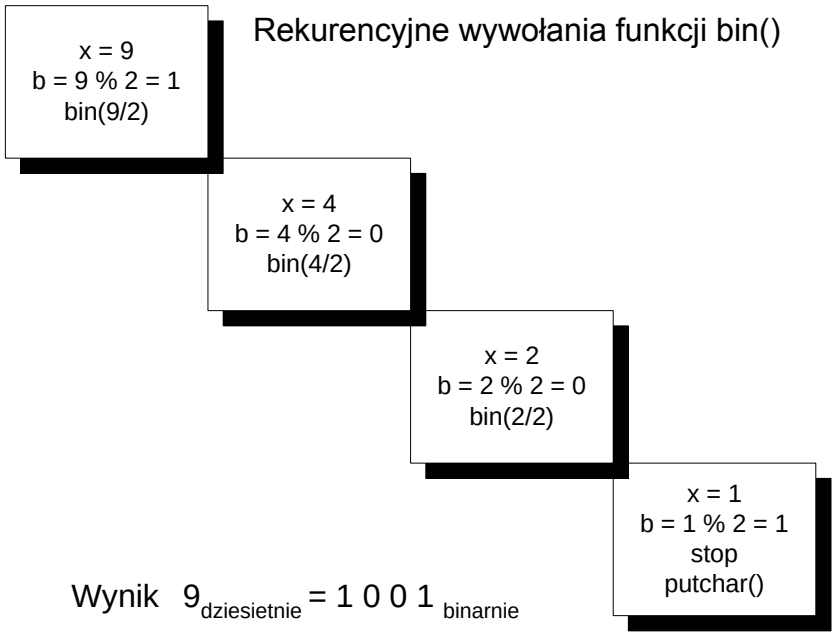
Zamiana liczby w systemie dziesiętnym na zapis binarny polega na rozłożeniu liczby dziesiętnej na sumę potęg liczby 2 ( $b_i\ 2^i$ +.....+ $b_1\ 2^1$ + $b_0\ 2^0$ ; $b_i$ może mieć wartość 0 lub 1). Praktyczny algorytm polega na iteracyjnym dzieleniu liczby dziesiętnej x przez 2, reszta z dzielenia jest notowana, a wynik ponownie jest dzielony przez dwa, itd. Liczby nieparzyste w systemie dwójkowym kończą się cyfrą 1, a parzyste cyfrą 0. Jeżeli x jest liczbą, to ostatnią cyfrę w zapisie binarnym otrzymamy wykonując operację dzielenia modulo: x % 2. Jeżeli przetwarzana liczba będzie 9 (w zapisie binarnym: 1001) to mamy:

```
9 % 2 = 1
```

Ostatnia cyfra zapisu binarnego jest równa 1. Pierwsza obliczona cyfra jest ostatnią cyfrą w zapisie binarnym. Następnie wywoływana jest funkcja rekurencyjna z parametrem (x / 2):

```
bin(x / 2);
```

i operacja wyznaczania zapisu binarnego przeprowadzana jest rekurencyjnie. Przebieg obliczeń pokazany jest na rysunku 12.3.




Rys.12.3 Konwersja liczby dziesiętnej na liczbę binarną

W programie wyrażenie ( '0' + b ) ma wartość znaku '0', jeśli b jest równe 0 lub znaku '1' jeśli b wynosi 1. Pamiętamy, że obliczenie 1 % 2 daje wynik równy 1.

## 12.5. Bitowe operatory logiczne

Stosując operatory bitowe warto pamiętać reguły łączenia bitów zdefiniowane dla poszczególnych operatorów.

**Bitowa koniunkcja (AND) – operator &**

Operator bitowy AND działa na dwóch elementach tego samego typu, jest operatorem dwuargumentowym.

Reprezentacja bitowa obu elementów jest porównywana bit po bicie, analizie podlegają jedynie bity umieszczone na tych samych pozycjach, w wyniku powstaje trzeci element – wartość wyjściowa. Dany bit w wartości wyjściowej jest równy 1 tylko wtedy, gdy oba odpowiadające mu bity w operandach są równe 1. W tabeli podane są reguły łączenia bitów za pomocą operatora AND.

| Bit 1 | Bit 2 | Wynik |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Przykładem operacji bitowych niech będzie działanie operatora &. Niech dane x1 i x2 będą reprezentowane przez 8 bitów. Prześledzimy na przykładzie wykonanie operacji x1 & x2, co da w wyniku daną x3:

```
          x1 =        1 0 1 0 1 0 1 1
          x2 =        1 1 1 1 0 0 1 0
    ----------------------------------------
x3 = x1 & x2 =        1 0 1 0 0 0 1 0
```

**Bitowa alternatywa (OR) – operator |**

Operator bitowy OR działa na dwóch elementach tego samego typu, jest operatorem dwuargumentowym. Reprezentacja bitowa obu elementów jest porównywana bit po bicie, analizie podlegają jedynie bity umieszczone na tych samych pozycjach, w wyniku powstaje trzeci element – wartość wyjściowa. Dany bit w wartości wyjściowej jest równy 1, gdy przynajmniej jeden z odpowiadających mu bitów w operandach jest równy 1. W tabeli podane są reguły łączenia bitów za pomocą operatora OR.

| Bit 1 | Bit 2 | Wynik |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

W następującym przykładzie:

(10010011) | (00111101) == ( 10111111)

w wartości wynikowej tylko bit szósty ma wartość 0.

**Bitowa różnica symetryczna (XOR) – operator ^**

Operator bitowy XOR działa na dwóch elementach tego samego typu, jest operatorem dwuargumentowym. Reprezentacja bitowa obu elementów jest porównywana bit po bicie, analizie podlegają jedynie bity umieszczone na tych samych pozycjach, w wyniku powstaje trzeci element – wartość wyjściowa.
Dany bit w wartości wyjściowej jest równy 1, gdy dokładnie jeden z odpowiadających mu bitów w operandach jest równy 1. W tabeli podane są reguły łączenia bitów za pomocą operatora XOR.

| Bit 1 | Bit 2 | Wynik |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

W następującym przykładzie:

(10010011) ^ (00111101) == ( 10101110)

w wartości wynikowej bit zerowy ma wartość, 0 ponieważ bity na pozycji zerowej obu operandów na tej pozycji maja wartość 1. Bity 1, 2, 3, 5 i 7 mają różne wartości w obu operandach.

**Bitowa negacja – operator ~**
Operator bitowej negacji (dopełnienie jedynkowe) jest operatorem jednoargumentowym. Operator negacji działając na argument w reprezentacji bitowej zmienia każde zero na jedynkę i każdą jedynkę na zero.
W następującym przykładzie:

~(10001010) == ( 01110101)

w wartości wyjściowej zamienione zostały wszystkie wartości bitów.

**Bitowe operatory łączone**
Język C udostępnia także łączone operatory operacji bitowych:

| Operator | symbol | przykład | znaczenie |
|---|---|---|---|
| koniunkcja-przypisanie | &= | num &= 0xff | num = num & 0xff |
| alternatywa-przypisanie | \|= | num \|= 0xff | num = num \| 0xff |
| alternatywa wyłączająca – przypisanie | ^= | num ^= 0xff | num = num ^ 0xff |

**Testowanie operatorów bitowych**

Program do testowania operatora AND może mieć postać pokazaną na listingu 12.2. Dość wygodnym formatem dla danych wejściowych jest zapis szesnastkowy, ponieważ każda cyfra szesnastkowa reprezentuje dokładnie 4 bity.
Po uruchomieniu programu możemy mieć następujący wynik:

```
Podaj dwie liczby szesnastkowe: 1 0
01 & 00 = 00
Podaj dwie liczby szesnastkowe: 1 1
01 & 01 = 01
Podaj dwie liczby szesnastkowe: c 7
0c & 07 = 04
Podaj dwie liczby szesnastkowe: 0 0
00 & 00 = 00
```

Listing 12.2. Operacje bitowe, operator AND

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

int main()
{
 unsigned int a = 0xff, b = 0xff;
 while (a != 0 || b != 0)
  {printf("\nPodaj dwie liczby szesnastkowe: ");
   scanf("%x %x", &a, &b);
   printf("%02x & %02x = %02x\n", a, b, a & b);
  }
 getche();
 return 0;
}
```

Zanalizujemy efekt operacji & na szesnastkowych liczbach c i 7, korzystając z reprezentacji bitowej:

```
  c  = 1100
  7  = 0111
------------------
c & 7 =  0100  = 4
```

Operator bitowy AND jest używany do testowania, czy dany bit zmiennej jest ustawiony na 1 czy na 0.
Program do testowania działania bitowego operatora OR może mieć postać pokazaną na kolejnym listingu (12.3).

Listing 12.3. Operacje bitowe, operator OR

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

int main()
{
  unsigned int a = 0xff, b = 0xff;
  while (a != 0 || b != 0)
    {
     printf("\n Podaj dwie liczby szesnastkowe: ");
     scanf("%x %x", &a, &b);
     printf("%02x | %02x = %02x\n", a, b, a | b);
    }

  getche();
  return 0;
}
```

Po uruchomieniu programu możemy mieć następujący wynik:

```
Podaj dwie liczby szesnastkowe: 0 1
00 | 01 = 01
Podaj dwie liczby szesnastkowe: 1 1
01 | 01 = 01
Podaj dwie liczby szesnastkowe: c 7
0c | 07 = 0f
Podaj dwie liczby szesnastkowe: 0 0
00 | 00 = 00
```

Bitowy operator OR jest używany do łączenia bitów dwóch różnych zmiennych w jedną zmienną.

## 12.6. Bitowe operatory przesunięcia

Bardzo użyteczne są dwa operatory działające na pojedynczej zmiennej – operatory przesunięcia w prawo i przesunięcia w lewo. Operator przesunięcia w lewo (oznaczony symbolem << ) przesuwa każdy bit zmiennej w lewo o zadaną liczbę miejsc. W tabeli podano kilka przykładowych operacji.

| char x = ‘z’ | | |
|---|---|---|
| Wyrażenie | Reprezentacja bitowa | Wynik |
| X | 01011010 | Bez przesunięcia |
| X<<1 | 10110100 | Przesuniecie o 1 miejsce |
| X<<4 | 10100000 | Przesunięcie o 4 miejsca |

Podczas przesuwania w lewo, bity przesuwane są w obrębie 8 pól (dla zmiennej zapisanej na 1 bajcie), skrajny lewy bit wypada z sekwencji, miejsca z prawej strony uzupełniane są zerami. Podobnie działa operator przesunięcia w prawo – z tym tylko, że może występować problem ze znakiem (pierwszy bit z lewej strony). Przesunięcie o jeden bit w prawo jest równoważne podzieleniu zmiennej przez dwa, przesuniecie o jeden bit w lewo jest równoważne pomnożeniu zmiennej przez dwa.
Stosując operatory przesunięcia należy pamiętać o dwóch ograniczeniach – prawy operand nie może być ujemny oraz jego wartość nie może być większa niż liczba bitów reprezentujących lewy operand. Jeżeli te ograniczenia zostaną złamane, wynik jest nieokreślony.
W języku C mamy dwa operatory przesunięcia – przypisania:

>>= operator przesunięcia w prawo i przypisani
<<= operator przesunięcia w lewo i przypisania

Program pokazany na listingu 12.4 ilustruje wykorzystanie operatorów przesunięcia bitowego.

Listing 12.4. Operacje bitowe, operatory przesunięcia << i >>

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

int main()
{
 int a, b;
 int x = 8;
 a = x >> 1;
 b = x << 1;
 printf("\n x = % d a = %d b = %d", x,a,b);
 x >>= 2;
 printf("\nbity w x przesuniete o 2 miejsca w prawo");
 printf("\n x = %d", x);
 getche();
 return 0;
}
```

Po uruchomieniu programu możemy mieć następujący wynik:

```
x = 8 a = 4 b = 16
bity w x przesuniete o 2 miejsca w prawo
x = 2
```

Pewne komplikacje występują podczas operacji przesuwania bitów w prawo. Bity wykraczające poza prawa granicę pierwszego operandu są usuwane. Natomiast z lewej strony jest zwalniane miejsce, które musi być wypełnione. W przypadku wartości bez znaku, zwalniane miejsca są wypełniane zerami. W przypadku wartości ze znakiem, wynik działania operatora przesuwania bitów zależy od komputera: zwalniane miejsca mogą być wypełniane zerami lub kopiami bitu przechowującego znak.

## 12.7. Maski bitowe

W operacjach bitowych bardzo użyteczne są tzw. **maski.** Maska jest to stała (może być także zmienna), która służy do otrzymania wartości konkretnego bitu w zmiennej. Dla przypomnienia, 16 bitowa reprezentacja jedynki typu **int** ma postać:

```
0000000000000001
```

Taka maska może być zastosowana do określenia niższych bitów w wyrażeniu typu **int**. Pokazany niżej fragment kodu używa maski i drukuje sekwencje zer i jedynek:

```
int k;
for (k = 0;  k < 10; ++k)
  printf(”%d”,k & 1);
```

Jeżeli zastosujemy stałą szesnastkową 0x10, to otrzymamy maskę dla piątego bitu (licząc z prawej strony).
Wyrażenie:

```
(x & 0x10) ? 1 : 0
```

ma wartość 1 lub 0 w zależności od wartości piątego bitu w zmiennej **x.** Reprezentacja bitowa wartości 0x10 ma postać: 00010000.
Wyrażenie:

```
(x & 0x20) ? 1 : 0
```

ma wartość 1 lub 0 w zależności od wartości szóstego bitu w zmiennej **x.** Reprezentacja bitowa wartości 0x20 ma postać: 00100000. Możemy zauważyć, że kod dwójkowy dużej litery na pozycji szóstego bitu (licząc od prawej) ma wartość 0, a dla małej litery ma wartość 1.
W wielu sytuacjach manipulowanie bitami jest bardzo przydatne. Przypuśćmy, że zdefiniowaliśmy stałą symboliczną MASKA, jako liczbę 2, czyli binarnie, jako (00000010). Zastosowanie instrukcji:

```
num  = num & MASKA;
```

spowoduje przypisanie wartości 0 wszystkim bitom zmiennej **num** z wyjątkiem bitu numer 1 (przypominamy, że numeracja bitów zaczyna się od zera z lewej strony), ponieważ każdy bit połączony z zerem za pomocą operatora & daje 0. Często stosuje się maskę 0xff - w systemie binarnym jest to (11111111). Instrukcja:

```
zn &= 0xff  (równoważna postać zn = zn & oxff)
```

spowoduje, że w zmiennej **zn** ostatnie 8 bitów pozostaje bez zmian, natomiast pozostałe bity zostaną wyzerowane. Niezależnie czy zmienna **zn** ma rozmiar 1,2 lub więcej bajtów, zostanie ona skrócona do jednego bajta.
Często zachodzi potrzeba włączenia określonych bitów w reprezentacji binarnej wartości, przy jednoczesnym pozostawieniu reszty bez zmian. W sterowaniu mamy często do czynienia z sytuacją, że jakieś urządzenie jest włączone (1) lub wyłączone (0), musimy zmieniać wartość tylko jednego bitu.
Dla stałej MASKA (MASKA = 2), w której tylko bit nr 1 ma wartość 1 dzięki instrukcji:

```
num |= MASKA  (równoważna postać num = num |MASKA)
```

nadana zostanie wartość 1 bitowi nr 1 w zmiennej **num.**
Wyłączanie bitu możemy zrealizować za pomocą negacji stałej MASKA:

```
num = num ~MASKA
```

Gdy MASKA ma wartość 2, to składa się z samych zer z wyjątkiem bitu nr 1, który jest jedynką. Korzystając z operatora negacji ~MASKA, otrzymamy w masce same jedynki z wyjątkiem bitu nr 1, który będzie zerem.

Aby odwrócić bit w wartości należy zastosować bitowy operator alternatywy wyłączającej (XOR). Połączenie wartości zmiennej z maską za pomocą operatora XOR spowoduje odwrócenie tych bitów, które w masce są włączone i pozostawienie bez zmian pozostałych. Aby odwrócić bit nr 1 w zmiennej **num** należy użyć instrukcji (MASKA = 2):

```
num = num ^ MASKA
```

Aby sprawdzić czy bit nr 1 w zmiennej **num** jest włączony należy zastosować instrukcję:

```
if (num & MASKA) == MASKA
   printf(”Jest”):
```

Należy najpierw zamaskować pozostałe bity w zmiennej **num** a potem porównać bit nr 1.
Jak wiemy, konwersję z kodu dziesiętnego na szesnastkowy i odwrotnie, możemy wykonać korzystając z kodu formatującego **%x** w funkcji **printf()** i **scanf().** Nie dysponujemy prostym mechanizmem, aby dokonać konwersji z systemy szesnastkowego na binarny. Przy zastosowaniu masek bitowych możemy opracować taki program (program pokazany na listingu 12.5).

Listing 12.5. Konwersja liczby szesnastkowej na binarną

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

int main()
{
  int num = 0xff, bit;
  unsigned int maska;
  printf("\nhex -> bin, zero konczy program");
  while (num != 0)
    {  maska = 0x8000;
       printf("\n Podaj liczbe: ");
       scanf("%x", &num);
       printf("Liczba %04x binarnie = ",num);
       for (int i = 0; i < 16; i++)
        { bit = (maska & num) ? 1 : 0;
          printf("%d " , bit);
          if (i == 7)     printf("-- ");
          maska >>=1;
        }
  }
  getche();
  return 0;
}
```

Po uruchomieniu programu (listing 12.5) możemy otrzymać następujący wydruk:

```
Hex -> bin, zero konczy program
Podaj liczbe :  1
Liczba 0001 binarnie = 0 0 0 0 0 0 0 0 - 0 0 0 0 0 0 0 1
Podaj liczbe :  80
Liczba 0080 binarnie = 0 0 0 0 0 0 0 0 - 1 0 0 0 0 0 0 0
Podaj liczbe :  100
Liczba 0100 binarnie = 0 0 0 0 0 0 0 1 - 0 0 0 0 0 0 0 0
Podaj liczbe :  ff
Liczba 00ff  binarnie = 0 0 0 0 0 0 0 0  - 1 1 1 1 1 1 1 1
Podaj liczbe :  0
Liczba 0000 binarnie = 0 0 0 0 0 0 0 0 - 0 0 0 0 0 0 0 0
```

Zasadnicza część programu wykonywana jest w pętli **for:**

```
for (int i = 0; i < 16; i++)
     {  bit = (maska & num) ? 1 : 0;
        printf("%d " , bit);
        if (i == 7)     printf("-- ");
        maska >>=1;
     }
```

W pętli **for** badane są wszystkie bity zmiennej całkowitej, zaczynając od skrajnego lewego. W instrukcji:

```
bit = (maska & num) ? 1 : 0;
```

wykorzystano maskę. Przy pierwszym przebiegu pętli, maska będzie miała postać binarną 10000000 00000000. Po wykonaniu operacji AND na masce i zmiennej **num** mamy informację o bicie. Jeżeli rezultat jest niezerowy (prawda), wiemy, że pierwszy bit w zmiennej **num** jest 1, w przeciwnym przypadku jest on 0. Zmienna **bit** otrzymuje odpowiednią wartość.
Po wydrukowaniu wartości, wykonywana jest instrukcja:

```
maska >>=1;
```

która przesuwa bity w masce o jedno miejsce w prawo. Po drugim przebiegu maska ma postać: 01000000 00000000. Ta zmodyfikowana maska służy do określenia kolejnego bitu. W końcu zostaną wyznaczone wszystkie bity badanej liczby.
W kolejnym przykładzie pokażemy wykorzystania operacji bitowych do obsługi znaków. Większość komputerów posługuje się kodami ASCII. Znakom przyporządkowane są wartości (kody znaków). Np. znakowi A odpowiada kod 65, znakowi a odpowiada kod 97, znakowi 1 odpowiada kod 49. Kody dziesiętne mają reprezentacje bitowe (liczby w systemie dwójkowym, wzorce bitowe). W normie ASCII bajt zawiera 8 bitów, numerowanych od 0 do 7.

Przyglądając się wzorcom bitowym możemy zauważyć, że jeżeli bajt 6 zawiera 1 (symbol jest literą), to bit 5 informuje, czy to litera duża( 0) czy mała (1).

Operacje bitowe AND, OR, XOR, NOT porównują dwa wzorce bitowe. Porównanie dotyczy bitów znajdujących się na tych samych pozycjach. Te wywody zilustrujemy przykładem operacji AND (w języku C oznaczana jest ta operacja symbolem &). Mama dwa bajty:

**Pozycja w bajcie : 7 6 5 4 3 2 1 0**
bajt1 : 0 1 1 0 1 0 0 1
bajt2 : 1 0 1 1 1 0 1 0

Przypominamy, że reguły łączenia bitów za pomocą operacji AND są następujące:

0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1

Łączenie naszych przykładowych bajtów operatorem **&** da w wyniku **bajt3**:

```
bajt1 & bajt2 = bajt3
```

Reprezentacja bitowa bajta wynikowego jest:

```
bajt3  :  0 0 1 0 1 0 0 0
```

Zauważmy, że jeżeli **bajt2** zawiera jedynkę na dowolnej pozycji to wykonanie operacji AND nie zmieni bitu w zmiennej **bajt1**, jeżeli występuje na jakiejś pozycji zero to po wykonaniu tej operacji otrzymujemy 0. Praktyczny przykład (program pokazany na listingu 12.6) pokazuje jak możemy używając znaków ASCII zmieniać litery duże na małe lub odwrotnie.

Listing 12.6. Operacje bitowe, maski, znaki

```
#include <stdio.h>
#include <string.h>    //dla strlen()
void main(void)
{ char tekst[50], dzn1,dzn2;
  char maska1 = 32;
  char maska2 = 223;
  int i,rozmiar;
  puts("\nEnter tekst :");
  gets(tekst);
  rozmiar = strlen(tekst);    //dlugosc lancucha
  for (i=0;i<rozmiar; i++)
  {dzn1 = tekst[i] ^ maska1; //alternatywa wylaczajaca
   dzn2 = tekst[i] & maska2;  // koniunkcja bitowa
   printf("\n%c %c %c ",tekst[i],dzn1,dzn2);
  }
 }
```

Po uruchomieniu programu z listingu 12.6 na ekranie pojawi się komunikat:

```
Enter tekst :
```

Jeżeli wykorzystując klawiaturę wpiszemy następujący tekst (po napisaniu tekstu naciskamy klawisz Enter):

```
To JEST tekst
```

W wyniku działania programu otrzymamy napis:

```
T    t    T
o    O    O
          
J    j    J
E    e    E
S    s    S
T    t    T
          
t    T    T
e    E    E
k    K    K
s    S    S
t    T    T
```

W kolumnie pierwszej widzimy tekst w postaci takiej, w jakiej został wprowadzony. W kolumnie drugiej program wypisał tekst zamieniając duże litery na małe oraz małe na duże. W kolumnie trzeciej, program przekształcił pierwotny tekst w taki sposób, że niezależnie od tego czy wpisywaliśmy z klawiatury litery duże czy małe, otrzymaliśmy tekst napisany literami dużymi. Ten program, oprócz wielu innych wad ma jedną istotną – w miejscu spacji pojawił się inny znak. W ramach ćwiczeń, proponujemy zidentyfikować problem i zaproponować rozwiązanie. Do wczytania tekstu wykorzystaliśmy funkcję biblioteczną **gets()**. Wprowadzanie łańcucha jest zakończone, gdy użytkownik naciśnie klawisz Enter. Argument funkcji **gets()** może być zadeklarowany w następujący sposób:

```
char tekst[rozmiar]
```

gdzie **rozmiar** jest rozmiarem tablicy, w naszym przypadku rezerwujemy tablicę do wprowadzenia 49 znaków. Funkcja biblioteczna **strlen()** służy do obliczenia ilości wprowadzonych znaków, zwracana wartość wykorzystana została w pętli **for**. Kluczowe znaczenie w naszym programie mają dwie instrukcji:

```
dzn1 = tekst[i] ^ maska1;  //XOR
dzn2 = tekst[i] & maska2;  //AND - koniunkcja bitowa
```

W zmiennej **dzn1** przechowujemy wynik działania operatora bitowego XOR. Obliczana jest alternatywa wyłączająca (XOR) znaku przechowywanego w zmiennej **tekst[i]** i zmiennej **maska1**. Załóżmy, że w tej zmiennej przechowujemy znak T.

Kod binarny litery **T** ma postać:

$$T : 84_{\text{kod dziesiętny}} : 0\,1\,0\,1\,0\,1\,0\,0$$

Wiemy także, że kod binarny litery **t** ma postać:

$$t : 116_{\text{kod dziesiętny}} : 0\,1\,1\,1\,0\,1\,0\,0$$

Reprezantacją bitową liczby (dziesiętnej) 32 jest:

$$32 : 0\,0\,1\,0\,0\,0\,0\,0$$

Zatem operacja T XOR 32 ma postać:

```
        0 1 0 1 0 1 0 0
        0 0 1 0 0 0 0 0
XOR :   0 1 1 1 0 1 0 0
```

Otrzymany wzorzec bitowy odpowiada literze małe **t**. Podobnie otrzymujemy wynik działania operatora bitowego AND. Reprezantacją bitową liczby (dziesiętnej) 223 jest:

$$223 : 1\,1\,0\,1\,1\,1\,1\,1$$

Wykonujemy operację AND na tych bajtach:

```
        0 1 0 1 0 1 0 0
        1 1 0 1 1 1 1 1
AND :   0 1 0 1 0 1 0 0
```

W wyniku operacji na bitach otrzymaliśmy niezmienioną reprezentację bitową litery duże T, czyli znak ten nie będzie zmieniany.
Korzystając z maski o wartości 1, możemy opracować prosty program (listing 12.7), produkujący sekwencje na przemian zer i jedynek

Listing 12.7. Operacje bitowe, sekwencja zer i jedynek

```
#include <stdio.h>
#include <conio.h>
int main()
{
  for (int i = 0; i< 10; ++i)
    printf("%d", i & 1);
  getche();
  return 0;
}
```

Po uruchomieniu tego programu otrzymamy następującą sekwencję: 0101010101.
W kolejnym programie pokazanym na listing 12.8 wykorzystamy maski w programie drukującym reprezentację bitowa wartości typu **int**.

Listing 12.8. Operacje bitowe, reprezentacja bitowa

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

int main()
{int v = 32;
 printf("typ int ma rozmiar= %4d bajty\n", sizeof(v));
 printf("\nzmienna int v = %d ma reprezentacje\n\n");
 int maska = 1;              //maska  000......01
 maska <<= 31;               //przesuwa 1 bit to konca
 for (int i = 1; i <= 32; ++i)
   { putchar(((v & maska) == 0) ? '0' : '1');
     v <<=1;
   }
getche();
return 0;
}
```

Po uruchomieniu tego programu mamy wydruk:

```
Typ int ma rozmiar =     4 bajty
Zmienna int  v = 32 ma reprezentacje
00000000000000000000000000100000
```

Operowanie bitami ma wiele zalet między innymi pozwala na lepsze wykorzystanie baju, co nie tylko pozwala oszczędzać pamięć, ale obliczenia mogą też być wykonywane szybciej. Jeżeli maszyna przetwarza 2 bajty instrukcji w cyklu, jest to równoważne przetwarzaniu 16 bitów równolegle (jednocześnie). Podobnie, gdy przetwarzane są 4 bajty w jednym cyklu to jest to równoważne przetwarzaniu 32 bitów równolegle. Zysk na czasie wykonywania programu może być oczywisty.
Przedstawiony program (listing 12.9) pokazuje, w jaki sposób możemy umieścić dwie zmienne znakowe (każda potrzebuje 8 bitów) w zmiennej typu **int** (16 bitów). Pakowanie znaków do zmiennej typu **int** realizowane jest w instrukcjach:

```
//pakowanie
  p = c1;
  p = (p<<8) | c2;
```

W zmiennej **p** typu **int** umieszczona dwa znaki A i B. Wykorzystano najpierw operator przesunięcia bitów o osiem miejsc w lewo a następnie operator bitowy OR.
Oczywiście pakowanie danych ma sens, jeżeli możemy potem te dane odzyskać.

Możemy operacje rozpakowywania zrealizować za pomocą instrukcji:

```
//rozpakowanie
  x1 = p & 0xff;
  x2 = (p & 0xff00) >> 8;
```

Wykorzystano w tym celu operator bitowy AND.

Listing 12.9. Operacje bitowe – kompresja znakow, 16 bitów

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

int main()
{
 char c1, c2, x1, x2;
 int p;
 c1 = 'A';
 c2 = 'B';
 printf("\nzmienna c1 = %c, zmienna c2 = %c", c1, c2);
 //pakowanie
        p = c1;
        p = (p<<8) | c2;
 //rozpakowanie
        x1 = p & 0xff;
        x2 = (p & 0xff00) >> 8;

 printf("\nzmienna upakowana x1 = %c", x1);
 printf("\nzmienna upakowana x2 = %c", x2);
 getche();
 return 0;
}
```

Po uruchomieniu tego programu mamy następujący komunikat:

```
zmienna c1 = A, zmienna c2 = B
zmienna upakowana x1 = B
zmienna upakowana x2 = A
```

W nowoczesnych systemach komputerowych dominują procesory 32 bitowe. W takiej sytuacji zmienna typu **int** ma rozmiar 4 bajtów, dzięki czemu możemy do niej zapakować cztery zmienne typu **char**. W kolejnym programie, pokazanym na listingu 12.10 , pokażemy ten proces, wykorzystamy także osobna funkcje do drukowania reprezentacji binarnej upakowanych znaków. W tablicy znakowej elementami są cztery litery: a,b,c i d. Te cztery znaki zostaną zapakowane do 32 bitowej zmiennej **p** typu **int**. Na wydruku otrzymujemy reprezentacje bitową zmiennej **p** i bity przyporządkowane znakom.

Listing 12.10. Operacje bitowe – kompresja znaków, 32 bity

```
#include <stdio.h>
#include <conio.h>
void drukuj_bity(int);

int main()
{char c[] = {'a','b','c','d'};
 int p;
      //pakowanie znakow
 p = c[0];
 p = (p << 8) | c[1];
 p = (p << 8) | c[2];
 p = (p << 8) | c[3];
 drukuj_bity(p);
      //rozpakowanie znakow
 c[3] =  p & 0xff;
 c[2] = (p & 0xff00) >> 8;
 c[1] = (p & 0xff0000) >> 16;
 c[0] = (p & 0xff000000) >> 24;

 putchar('\n');
 for (int j = 0; j < 4; ++j)
          printf(" %8c",c[j]);

  getche();
  return 0;
}

void drukuj_bity(int v)
{ //drukuje bity reprezentujace dane typu int
   int maska = 1;      //maska  000......01
   maska <<= 31;       //przesuwa 1 bit to konca
   putchar('\n');
   for (int i = 1; i <= 32; ++i)
     { putchar(((v & maska) == 0) ? '0' : '1');
       v <<=1;
       if (!(i % 8)) putchar (' ');
     }
}
```

Po uruchomieniu programu mamy następujący wydruk:

```
01100001 01100010 01100011 01100100
    a         b      c        d
```

## 12.8. Unie

Unia (ang. *union*) jest strukturą, której wszystkie składowe umieszczane są w tym samym obszarze pamięci. Kompilator ustala ilość potrzebnej pamięci wyliczając ilość pamięci potrzebnej do przechowania składowej o największym zapotrzebowaniu. W dowolnej chwili unia może zawierać maksymalnie jeden obiekt (składową), ponieważ elementy unii dzielą ten sam obszar pamięci.

Deklaracja unii jest podobna do deklaracji struktury, zamiast słowa kluczowego ***struct*** występuje słowo ***union***. Przykładowa deklaracja unii (szablon unii z etykietą) może mieć postać:

```
union zmien
{
   int x1;
   char zn;
};
```

Zmienne typu unii można deklarować przez ich nazw na końcu definicji unii albo w oddzielnej instrukcji deklaracji. Dostęp do składowych unii jest analogiczny jak dla struktur.

Aby zadeklarować zmienną **wx1**, jako typu **zmien** należy napisać:

```
union zmien wx1;
```

Zarówno liczba całkowita **x1** jak i znak **zn** zmiennej **wx1** zajmują ten sam obszar pamięci ( zazwyczaj potrzebujemy dwóch bajtów dla zmiennej typu **int** i jednego bajta dla zmiennej typu **char**).

Jeżeli mamy zdefiniowany szablon unii z etykietą **zmien**, legalne definiowanie unii może mieć postać:

```
union zmien x1          // unia typu zmien
union zmien tab[100]    //tablica 100 unii typu zmien
union zmien *y1         //wskaźnik do unii typu zmien
```

W pierwszej deklaracji utworzono pojedynczą zmienną o nazwie **x1**. Została przydzielona taka ilość pamięci, aby zmienna mogła przechowywać największą ze zmiennych wchodzących w skład szablonu unii **zmien** – w naszym przykładzie jest to zmienna typu **int,** jej rozmiar określić możemy, jako **sizeof(int).** Następna deklaracja tworzy tablicę o nazwie **tab**, która ma 100 elementów, każdy ma rozmiar **sizeof(int).** Ostatnia deklaracja tworzy wskaźnik, który może przechowywać adres unii typu **zmien**. Unie umożliwiają oszczędne gospodarowanie pamięcią komputera.

Inicjalizując unie należy pamiętać, że unia może przechowywać tylko jedną wartość w danym momencie. W języku C istnieją dwa sposoby inicjalizacji unii: nadanie unii wartości innej unii tego samego typu lub inicjalizacja pierwszego składnika unii.

Poniższy fragment kodu:

```
union {
       int x;
       float y;
       char c[10]
}      moja_unia {33}
```

powoduje, że składnik **x** (jest to pierwszy element ) unii o nazwie **moja_unia** został zainicjalizowany wartością 33.
W programie pokazanym na listingu 12.11 pokazano użycie unii. Zostały zadeklarowane: jedna struktura i jedna unia, każda z nich ma takie same składowe. Program podaje ilość pamięci, jaki zajmuje struktura i unia. W programie pokazano pewne niuanse stosowania unii. Niezbyt ostrożne stosowanie unii może być potencjalnym źródłem błędów, co zostało zilustrowane w omawianym programie.

Listing 12.11 Struktury i unie

```
#include <stdio.h>
#include <conio.h>
#include <string.h>     // dla strcpy()
int main(int)
{ struct wx1
   { int tel;
     char nazwisko[50];
     char miasto[50];
   };
  union wx2
   { int tel;
     char nazwisko[50];
     char miasto[50];
   };

  struct wx1 x1;
  union wx2 x2;
  printf("\nrozmiar struktury=%d bajtow", sizeof(x1));
  printf("\nrozmiar unii    = %d bajtow", sizeof(x2));
  strcpy(x2.nazwisko,"Kowalski");
  printf("\n x2.nazwisko = %s",x2.nazwisko);
  x2.tel = 8823;
  printf("\n x2.nazwisko = %s",x2.nazwisko);
  getche();
  return 0;
}
```

Ten program wyprowadza na ekran następujące linie:

```
rozmiar struktury    =  100 bajtow
rozmiar unii         =   50 bajtow
x2.nazwisko = Kowalski
x2.nazwisko = w”owalski
x2.nazwisko = Lublin
```

Widać ewidentną oszczędność pamięci. W obszarze pamięci rezerwowanej na składowe unii następuje nakładanie się danych, co zilustrowane w drugim wydruku składowej **x2.nazwisko** – zniekształcenie dotyczy dwóch pierwszych bajtów.

## 12.9. Pola bitowe

Język C umożliwia określenie liczby bitów, w których będą zapamiętywane składowe typu **unsigned** lub **int** struktury, klasy lub unii. Składowe takie nazywane są ***polami bitowymi*** ( ang. *bit field*)**.** Typem pola bitowego musi być liczba całkowita – ze znakiem lub bez. Pól bitowych używa się w celu zaoszczędzenia pamięci. Bardzo często w programach musimy mieć dane reprezentujące wartości binarne. Przykładem jest reprezentacja stanu konkretnego przełącznika – przełącznik może być włączony lub wyłączony. Do opisania działania przełącznika potrzebujemy jednego bitu (wartość 0 lub 1). Najmniejszym standardowym typem danych jest typ **char**. Używanie typu **char** do przechowywania wartości binarnych jest dużym marnotrawieniem pamięci. Zwykle jednak używa się typu **int,** który może składać się z dwóch lub częściej z czterech bajtów. W wielu przypadkach do reprezentowania wartości zmiennej możemy potrzebować niewielkiej ilości bitów (2, 3). Możemy zebrać kilka niedużych zmiennych razem, jako pola struktury. Składową struktury definiuje się, jako pole przez podanie jej nazwy, a za nią liczby potrzebnych bitów. Używając pól możemy zmieścić w typie **char** osiem wartości binarnych, a typie **long** trzydzieści dwie wartości. Na przykład, poniższa deklaracja tworzy strukturę **pol_bit** zawierającą trzy pola o rozmiarze 1 bitu:

```
struct {
        unsigned int nra  : 1;
        unsigned int nrb  : 1;
        unsigned int nrc  : 1;
} pol_bit;
```

Poszczególnym polom przypisanie wartości początkowych odbywa się klasycznie, z wykorzystaniem operatora kropki:

```
pol_bit.nra = 1;
pol_bit.nrb = 0;
pol_bit.nrc = 1;
```

W tym przykładzie, każde użyte pole ma długość 1 bitu, możemy, więc przypisywać jedynie wartości 0 lub 1.

Wykorzystanie pamięci nie jest imponujące w tym przykładzie. Zmienna **pol_bit** jest typu **int**, rezerwuje, więc 16 (lub 32 w zależności od implementacji) bitów, gdy tymczasem wykorzystywane są jedynie trzy bity.
Pola bitowe mogą mieć różna długość:

```
Struct {
          unsigned int nra  : 2;
          unsigned int nrb  : 2;
          unsigned int nrc  : 8;
} pol_bit;
```

W tym przykładzie mamy dwa pola 2-bitowe i jedno 8-bitowe. Dysponując szerszymi polami, możemy operować większymi wartościami, np. teraz zmienna **nra** może być zainicjalizowana wartością 3. Musimy jednak pamiętać, aby przypisując wartości nie przekroczyć pojemności pola. Jeżeli całkowita ilość bitów przypisana poszczególnym polom przekroczy rozmiar typu **int**, rezerwowana jest kolejna jednostka o długości typu **int**. Ponieważ żadne z pól nie może znajdować się na granicy pomiędzy dwoma jednostkami, kompilator tak rozmieszcza pola, aby znalazły się całkowicie w pierwszej bądź drugiej jednostce. W razie potrzeby tworzone są puste obszary. Aby umieścić pusty obszar należy zdefiniować szerokość bez podawania nazwy:

```
struct {
          unsigned int nra  : 2;
          unsigned int nrb  : 2;
          unsigned int      : 4;
          unsigned int nrc  : 8;
} pol_bit;
```

W powyższej strukturze występuje 4-bitowa przerwa pomiędzy **pol_bit.nrb** i **pole_bit.nrc**. Użycie pola o szerokości 0 sprawia, że następne pole zostaje wyrównane do początku kolejnej jednostki typu **int.**
Wykorzystanie pól bitowych ilustruje program pokazany na listingu 12.12, który symuluje rozdanie pięciu kart. Zakładamy, że w talii mamy 52 karty. Mamy następujący zestaw kart: dwójka, trójka, czwórka, piątka, szóstka, siódemka, ósemka, dziewiątka, dziesiątka, walet, dama, król, as. Każda karta może należeć do jednego z typów: trefl, karo, kier, pik.
Po uruchomieniu programu z listingu 12.12 możemy otrzymać następujący wynik:

```
        2    kier
     dama    trefl
        3    trefl
        8    karo
        krol   trefl
  rozmiar zmiennej karta = 1 bajt
```

Dodatkowo, za pomocą operatora rozmiaru **sizeof()** otrzymaliśmy informacje ile bajtów zajmuje nasza struktura **karta.**

Losowaną kartę opisuje struktura **pok**:

```
struct pok {
             unsigned nr : 4;
             unsigned kol: 2;
};
```

Do opisu karty potrzebujemy dwóch pól: numer karty i kolor. Ponieważ mamy trzynaście kart (figur) w danym kolorze, do zapisu numeru karty ( od 0 do 12) wystarczą 4 pola bitowe, do zapisu koloru (trefl, karo, kier, pik) wystarczą 2 pola bitowe.

Listing 12.12 Struktury i pola bitowe

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>     //dla rand()
#include <time.h>       //dla srand(time(NULL))

struct pok {
             unsigned nr : 4;
             unsigned kol: 2;
};

int main()
{pok karta;
 int w;
 srand( time( NULL) );
 clrscr();
 char *kolor[4] = {"trefl","karo", "kier", "pik"};
 char *num[13] =
         {"as","2","3","4","5","6","7","8","9","10",
           "walet","dama","krol"};
 for (int i=0; i<5; i++)
  {karta.nr = rand() % 13;
   karta.kol= rand() % 4;
   printf("\n %8s %s",num[karta.nr],kolor[karta.kol]);
  }
 printf("\nrozmiar zmiennej karta=%d bajt",
            sizeof(karta));
 getch();
 return 0;
}
```

Składowa **nr** przechowuje wartości pomiędzy 0 (as) i 12 (król). Zauważmy, że 4 bity mogą przechowywać wartości pomiędzy 0 i 15. Składowa **kol** przechowuje wartości pomiędzy 0 i 3 ( 0 = trefl, 1 = karo, 2 = kier, 3 = pik).

Losowanie pięciu kart realizowane jest w pętli:

```
for (int i=0; i<5; i++)
   { karta.nr = rand() % 13;
     karta.kol= rand() % 4;
     printf("\n  %8s  %s",num[karta.nr],
              kolor[karta.kol]);
   }
```

Do losowania figury i kolor użyto ***generatora liczb pseudolosowych***. Funkcja **rand()** generuje liczbę całkowitą z przedziału 0 i **RAND_MAX**. Funkcja **rand()** znajduje sie w bibliotece standardowej, do programu musimy włączyć plik nagłówkowy **<stdlib.h>.** Przypisanie liczby losowej z zakresu 0 – 32767 realizuje się za pomocą instrukcji:

```
licz_los = rand();
```

Mówiąc opisowo, funkcja **rand()** symuluje losowanie liczb z urny (podobnie jak to się dzieje podczas losowania liczb w popularnej grze liczbowej). Wartość **RAND_MAX** musi być przynajmniej równa 32767 (maksymalna dodatnia wartość dla liczby dwubajtowej). Ponieważ chcemy generować liczby z zakresu 0 – 12, musimy dokonać tzw. skalowania, czyli ograniczyć zakres generowanych liczb do naszego przedziału. Wykorzystujemy do tego celu operator dzielenia modulo:

```
karta.nr = rand() % 13;
karta.kol= rand() % 4;
```

W pokazanej pierwszej instrukcji, składowa struktury **nr** przyjmuje wartość liczby losowej z przedziału 0-12, w drugiej pokazanej instrukcji składowa **kol** przyjmuje wartość z przedziału 0 - 3.
Jeżeli nasz program będzie uruchamiany wielokrotnie, to za każdym razem otrzymamy ten sam wynik. Aby losowanie za każdym razem było różne, należy użyć funkcji **srand().** Funkcja **srand()** pobiera argument całkowity typu **unsigned** i powoduje, że funkcja **rand()** po każdym uruchomieniu daje inny wynik. Wygodnym sposobem uruchamiania funkcji **srand()** jest instrukcja:

```
srand( time( NULL) );
```

Argument funkcji **srand()** jest otrzymywany z funkcji **time(NULL).** Funkcja **time(NULL**) zwraca bieżący „czas kalendarzowy” w sekundach. Ta wartość jest przekształcana do liczby całkowitej bez znaku i przekazywana, jako argument do funkcji **srand().** Funkcję **srand()** wywołujemy tylko raz w programie. Parametry wylosowanej karty drukuje funkcja **printf():**

```
printf("\n  %8s  %s",num[karta.nr],kolor[karta.kol]);
```

Aby wynik losowania kart był bardziej czytelny, zamiast losowanych wartości liczbowych, drukowane są napisy, takie jak:

```
dama   trefl
```

W rzeczywistości mamy do czynienia z indeksami, w pokazanym przykładzie wylosowano liczby 11 (dama) i 0 (trefl). Do drukowania napisów wykorzystaliśmy tablice łańcuchów, a konkretnie w naszym przykładzie zastosowaliśmy tablice wskaźników:

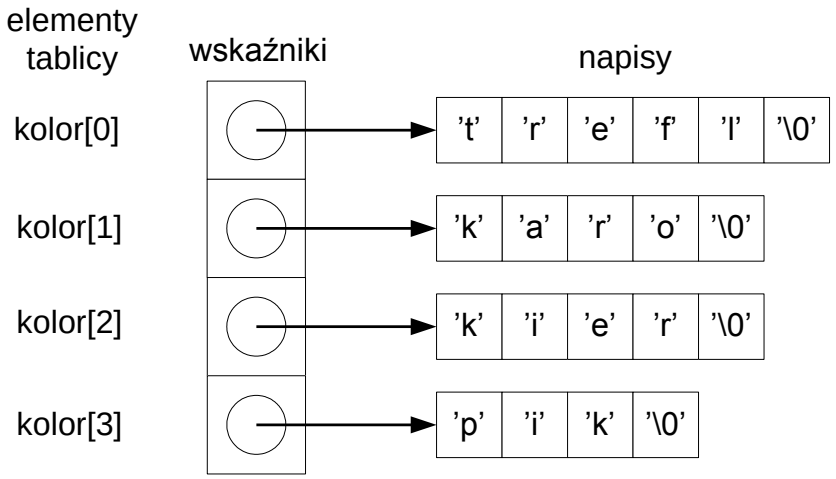
```
char *kolor[4] = {"trefl","karo", "kier", "pik"};
char *num[13] =
    {"as","2","3","4","5","6","7","8","9","10",
     "walet","dama","krol"};
```

Rozważmy deklaracje tablicy:

```
char *kolor[4] = {"trefl","karo", "kier", "pik"};
```

Deklaracja **kolor[4]** mówi, że mamy do czynienia z tablicą zawierająca cztery elementy. Napis **char *** oznacza, że mamy do czynienia ze wskaźnikiem do typu **char.** Elementami tej tablicy są napisy "trefl", "karo", "kier" oraz "pik". Tablica **kolor[ ]** zawiera wskaźniki do napisów. Każdy z nich wskazuje na pierwszy znak odpowiedniego napisu. Podobnie jest w przypadku tablicy **num[ ].** Graficzna reprezentacja tablicy **kolor[ ]** pokazana jest na rysunku 12.4. Tablica **kolor[ ]** ma stałą długość, ale dzięki niej możemy operować napisami o dowolnej długości. Jest to przykład elastyczności języka C.

Należy jednak zauważyć, że nasz program nie jest doskonały w tym sensie, że może się zdarzyć, że zostaną wylosowane dwie takie same karty. Ulepszony program powinien mieć mechanizmy zabezpieczające przed takim losowaniem. Każdorazowo należy sprawdzić, czy dana karta została już wylosowana, jeżeli tak, to należy powtórzyć losowanie.



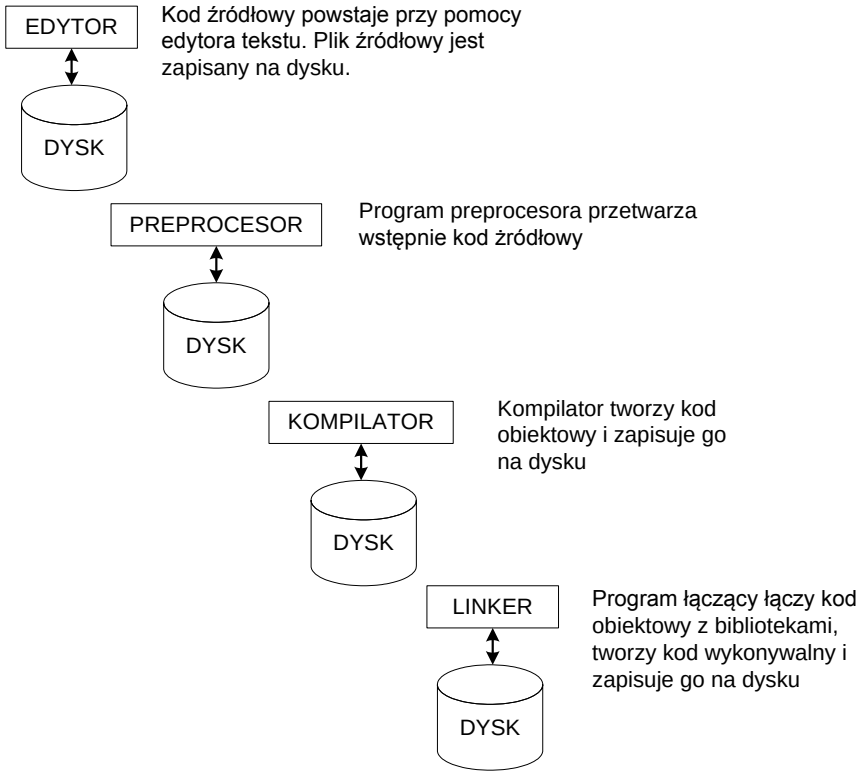

Rys.12.4. Graficzne przedstawienie tablicy wskaźników

# ROZDZIAŁ 13

## PREPROCESOR

## 13.1. Wstęp

Typowe środowisko C składa się z wielu części: środowiska projektowania programu, języka i biblioteki standardowej C. Typowe otoczenie projektowania i wykonywania programu w C pokazane jest na rysunku 13.1.




Rys. 13.1. Typowe środowisko C

Zanim na ekranie monitora otrzymany zostanie wynik działania programu komputerowego, program w C standardowo przejdzie przez pięć faz:

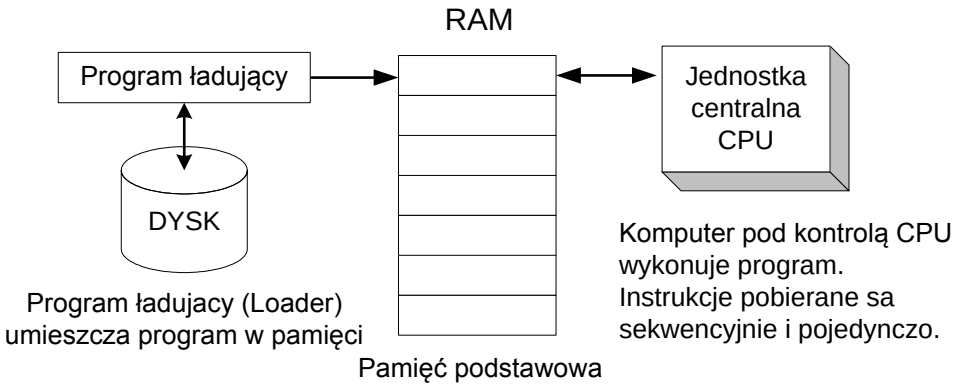
- edycja
- przetwarzanie wstępne
- kompilacja
- łączenie
- załadowanie i wykonanie

Napisany przez programistę, za pomocą edytora tekstu, program musi być ***skompilowany***. Kompilator tłumaczy program w C na kod języka maszynowego. Programista chcąc skompilować program wydaje środowisku programistycznemu polecenie kompilacji.

## 13.2. Dyrektywy preprocesora

Zanim środowisko wykona kompilację, uruchomiony zostaje ***program preprocesora*** – mamy do czynienia z fazą przetwarzania wstępnego. Uruchomienie programu preprocesora następuje automatycznie. Kody źródłowe języka C zawierają odpowiednie polecenia skierowane do preprocesora, noszą one nazwę ***dyrektyw preprocesora***.

W oparciu o dyrektywy preprocesor zmienia obecne w programie symboliczne skróty na ich definicje, dołącza wybrane pliki, określa, które części kodu są widoczne dla kompilatora. Wszystkie dyrektywy preprocesora zaczynają się znakiem #. Dyrektywy preprocesora nie są instrukcjami języka C wiec nie są zakończone średnikiem (;).




Rys.13.2. Wykonywanie programu

## 13.3. Dyrektywa preprocesora #include

Dyrektywa **#include** powoduje dołączenie pliku do programu źródłowego. Po znalezieniu i rozpoznaniu dyrektywy **#include** preprocesor odnajduje plik o podanej nazwie i dołącza kopię jego zawartość do pliku źródłowego w miejscu, w którym znajduje się dyrektywa. Dołączane pliki mogą znajdować się w dowolnych miejscach. Dyrektywa **#include** występuje w dwóch następujących formach:

```
#include <nazwa_pliku>
#include ”nazwa_pliku”
```

Jeżeli nazwa pliku zawarta jest w nawiasach ostrych, to preprocesor poszukuje pliku w jednym lub kilku standardowych katalogach systemowych. Jeżeli nazwa pliku zawarta jest w cudzysłowie, preprocesor przeszukuje najpierw katalog bieżący lub inny katalog podany wraz z nazwą a następnie przegląda katalogi standardowe.

Dyrektywa **#include** najczęściej stosowana jest do włączania standardowych plików nagłówkowych jak **stdio.h** czy **conio.h.** W dołączanych plikach zawarte są informacje, jakie potrzebne są kompilatorowi. W pliku **stdio.h** znajdują się definicje funkcji takich jak **printf()** czy **scanf().** Rozszerzenie **.h** jest zgodnie z tradycją dodawane do nazw plików nagłówkowych. Inne dołączane pliki mogą mieć dowolne rozszerzenia.

| Dyrektywa **#include** | opis |
|---|---|
| #include <math.h> | przeszukuje katalogi systemowe |
| #include ”planety.h | przeszukuje bieżący katalog roboczy |
| #include ”c:\P_C\pm13.h” | przeszukuje katalog P_C na dysku C |

Korzystanie z dyrektyw preprocesora niesie wiele korzyści. Jako przykład rozważymy tworzenie pliku nagłówkowego. W utworzonym pliku o nazwie pm13.h zapisano średnie odległości planet układu słonecznego od Słońca. Za pomocą innej dyrektywy preprocesora (#define) utworzono stałe symboliczne. Stałe symboliczne o nazwach planet reprezentują odległości planety od Słońca w tysiącach kilometrów.

Listing 13.1 pliki dołączane

```
/*pm13.h */

//srednia odleglosc Planeta - Slonce(tysiace km)
#define MERKURY 58000.0
#define WENUS 108000.0
#define ZIEMIA 150000.0
#define MARS 228000.0
#define JOWISZ  778000.0
#define SATURN 1426000.0
#define URAN 2868000.0
#define NEPTUN 4494000.0
#define PLUTON 5900000.0
#define KSIEZYC 385.0

#define VC 299793.0       // (km/sek)
//koniec pliku
```

Tworzenie takiego niezależnego pliku ma tą zaletę, że pozwala łatwo uaktualniać listę odległości, a także dopisywać nowe stałe i definicje. Ten plik został utworzony za pomocą edytora tekstowego i zapisany w katalogu o nazwie **P_C** na dysku **C:\.** Plik jest wykorzystywany jest przez program, który wylicza ile czasu potrzebuje promień świetlny oby przebyć odległość planeta – Słońce.

Listing 13.2. Pliki dołączane

```
/* preprocesor, #include */
#include <stdio.h>
#include <conio.h>
#include "c:\P_C\pm13.h"
int main()
{double planety[6] = {MERKURY, WENUS, ZIEMIA,
                      MARS, JOWISZ, SATURN};
 const char *pla[ ] = {"MERKURY", "WENUS", "ZIEMIA",
                       "MARS", "JOWISZ", "SATURN"};
 double czas;
 int n=-1;
 clrscr();
 printf("\n Swiatlo biegnie ze Slonca\n ");
 while(n++ < 5)
  {czas = planety[n]/VC;
   printf("\n       do planety %8s ",pla[n]);
   printf(" %8.0f sek\n",czas*1000.0);
  }
 getch();
 return 0;
}
```

Włączenie o nazwie pliku pm13.h realizowane jest w dyrektywie preprocesora:

```
#include "c:\P_C\pm13.h"
```

Oto dane wyjściowe otrzymane z programu:

```
Swiatlo biegnie ze Słońca
do planety         MERKURY            193 sek
do planety         WENUS              360 sek
do planety         ZIEMIA             500 sek
do planety         MARS               761 sek
do planety         JOWISZ            2595 sek
do planety         SATURN            4757 sek
```

## 13.4. Dyrektywy preprocesora #define, #ifdef i #ifndef

Dyrektywa **#define** tworzy ***stałe symboliczne*** i ***makroinstrukcje*** (makroinstrukcje zostały omówione w rozdziale dotyczącym funkcji). Dyrektywa może znajdować się w dowolnym miejscu kodu, a zawarta w niej definicja obowiązuje do końca pliku. Dyrektywa preprocesora obejmuje obszar od symbolu # do znaku końca linii. Długość dyrektywy jest ograniczona do jednego wiersza. Ponieważ jednak kombinacja lewy ukośnik - znak nowej linii jest traktowany, jako odstęp, zatem jedna dyrektywa może zajmować obszar kilku wierszy.

Z punktu widzenia preprocesora obszar ten stanowi jeden wiersz logiczny. Stosowanie dyrektywy **#define** jest bardzo powszechne i przyczynia się do zwiększenia czytelności kodu i ułatwia wprowadzanie zmian.
Konstrukcja dyrektywy **#define** jest następująca:

```
#define identyfikator tekst_zastępujacy
```

Każdy wiersz **#define** składa się z trzech części. Pierwszy element tworzy sama dyrektywa **#define**. Drugim elementem konstrukcji jest ustalany przez programistę identyfikator (albo po prostu skrót). Identyfikator często nazywany jest ***makrem*** lub ***zamiennikiem***. Trzecim elementem jest tekst zastępujący (rozwinięcie makra). Preprocesor ustala listę skrótów (analizując wszystkie wystąpienia z liniami zaczynającymi się od **#define**) i dokonuje przeglądu pliku źródłowego. Gdy znajdzie jeden ze zdefiniowanych skrótów w kodzie źródłowym, zastępuje skrót jego rozwinięciem. Rozważmy skrót PI:

```
#define PI 3.1415926
```

Preprocesor odnajdzie w kodzie źródłowym wszystkie napisy PI (jest to stała symboliczna) i zastąpi je wartością 3.1415926. Należy jednak pamiętać, że tekstem zastępującym jest wszystko, co występuje po prawej stronie skrótu. Dla przykładu, jeżeli dyrektywa będzie miała postać:

```
#define PI = 3.1415926
```

a w kodzie źródłowym mamy fragment postaci:

```
obwod = 2 * PI * r ;
```

to zastąpienie stałej symbolicznej przyjmie nieoczekiwaną postać:

```
obwod = 2 * =3.1415926 * r ;
```

Może być to przyczyną wielu subtelnych błędów logicznych i składniowych.
Dyrektywa **#define** jest używana w celu podniesienia czytelności kodu i zapewnienia przenaszalności programów. Bardzo często w programie używamy stałych takich jak π czy prędkość światła:

```
#define PI 3.1415926   //liczba Pi
#define E 2.71828      //podstawa logarytmu naturalnego e
#define C 299792.4562  //prędkość światła w km/sek
```

Możemy wprowadzić specjalne stałe:

```
#define EOF (-1) //typowa wartość końca pliku
#define TRUE 1   //prawda
#define MAXINT 2147483647 //największa liczba int
```

Bardzo często w programach są używane najrozmaitsze ograniczenia (np. liczba dopuszczalnych iteracji, tolerancja błędu, itp.):

```
#define PETLE 100       //maksymalna liczba iteracji
#define ROZ 100         //ustalony rozmiar tablicy
#define TOL 1.0e-6      //akceptowany błąd obliczeń
```

Bardzo często zachodzi potrzeba sprawdzenia czy tak określony symbol został zdefiniowany. Służą do tego dyrektywy

```
#ifdef    (ang. if defined, jeśli zdefiniowany
#ifndef   (ang. if not defined, jeśli nie zdefiniowany)
```

Zastosowanie dyrektyw **#ifdef** i **#ifndef** wymaga użycia kończącej dyrektywy **#endif**. Oprócz wymienionych już dyrektyw mamy jeszcze jedną dyrektywę **#else,** która może być wstawiona pomiędzy dyrektywy **#ifdef** i dyrektywę **#endif**. Program z listingu 13.3 ilustruje użycie omawianych dyrektyw. W programie zdefiniowano symbole:

```
#define WINDOWS
#define WERSJA 6
```

Pusta definicja taka jak:

```
#define WINDOWS
```

w zupełności wystarcza, aby dyrektywa **#ifdef** uznała nazwę WINDOWS za zdefiniowaną.

Listing 13.3. Dyrektywy #ifdef, #else, #endif

```
/* dyrektywy #ifdef, #else */
#include <stdio.h>
#include <conio.h>
#define WINDOWS
#define WERSJA 6

int main()
{printf("\n Sprawdzanie definicji");
 #ifdef WINDOWS
   printf("\n symbol WINDOWS zdefiniowany");
 #else
   printf("\n symbol WINDOWS nie zdefiniowany");
 #endif
 #ifndef WERSJA
   printf("\n symbol WERSJA nie zdefiniowany");
 #else
   printf("\nsymbol WERSJA zdefiniowany jako %d",
            WERSJA);
 #endif
 #ifdef WINDOWS_VER
   printf("\n symbol WINDOWS zdefiniowany");
 #else
   printf("\n symbol WINDOWS_VER nie zdefiniowany");
 #endif
 getche();
 return 0;
}
```

Natomiast nazwa WINDOWS_VER rzeczywiście nie została zdefiniowana i komunikat:

```
symbol WINDOWS_VER nie zdefiniowany
```

jest komunikatem oczekiwanym. Po uruchomieniu programu pokazanego na listingu 13.3 na ekranie mamy komunikat:

```
Sprawdzanie definicji
symbol WINDOWS zdefiniowany
symbol WERSJA zdefiniowany jako 6
symbol WINDOWS_VER nie zdefiniowany
```

**Kompilacja warunkowa**

Dyrektywy **#ifdef** oraz **#endif** są wykorzystywane bardzo często do warunkowego kompilowania instrukcji języka C. W programie z listingu 13.4 pokazano wykorzystanie powyższych dwóch dyrektyw preprocesora do testowania programu.

Listing 13.4. dyrektywy **#ifdef** oraz **#endif**

```
#include <stdio.h>
#include <conio.h>
#define TEST
int main()
{int suma = 0;
  for (int i =1; i<=5; i++)
    {suma += i*i;
     #ifdef TEST
     printf("\ni = %d   suma = %d",i, suma);
     #endif
    }
 printf("\nsuma = %d",suma);
 getche();
 return 0;
}
```

Po uruchomieniu mamy następujący wynik:

```
i = 1   suma =  1
i = 2   suma =  5
i = 3   suma =  14
i = 4   suma =  30
i = 5   suma =  55
suma =  55
```

Jeżeli dyrektywę umieścimy wewnątrz komentarza:

```
// #define TEST
```

i ponownie uruchomimy program to otrzymamy następujący komunikat:

```
suma =  55
```

Stosowanie kompilacji warunkowej jest bardzo przydatne przy testowaniu programów i usuwaniu błędów.

**Operator #**

W standardzie ANSI C, gdy chcemy, aby argument makra zawarty w łańcuchu został zastąpiony argumentem faktycznym możemy skorzystać z operatora **#**. Jego działanie zilustrujemy przykładem.

Listing 13.5. Operator #

```
/* preprocesor, operator # */
#include <stdio.h>
#include <conio.h>
#define PR(x) printf("kwadratem liczby " #x " jest \
          %d\n",((x)*(x)))
int main()
{int a = 3;
 PR(a);
 PR(3 + 5);
 getch();
 return 0;
}
```

W wyniku działania tego programu otrzymujemy komunikat:

```
kwadratem liczby a jest 9
kwadratem liczby 3 + 5 jest 64
```

W instrukcji **PR(a)** symbol **#x** zostaje zastąpiony łańcuchem **"a",** w następnej instrukcji **PR(3 + 5)** symbol **#x** zostaje zastąpiony łańcuchem **"3 + 5".**

## 13.5. Zestaw słów kluczowych preprocesora

Zestaw dyrektyw preprocesora jest bardzo szeroki, zależy też od implementacji kompilatora. Aby poznać wszystkie dyrektywy preprocesora konkretnego kompilatora języka C należy zapoznać się z jego opisem technicznym.

Najczęściej stosowane dyrektywy to:

```
#include
#define
#undef
#ifdef
#else
#endif
#ifndef
#if
#elif
#error
#pragma
#line
```

Dyrektywa **#include** włącza kopie określonego pliku. Dyrektywa **#define** jest używana do tworzenia stałych symbolicznych i makroinstrukcji. Dyrektywa **#undef** służy do wyłączania stałych symbolicznych i makroinstrukcji, konkretnie usuwa ich definicje. Dyrektywy **#ifdef, #else, #endif, #ifndef, #if, #elif** są wykorzystywane najczęściej do kompilacji warunkowych. Dyrektywa **#error** wyświetla zależny od implementacji komunikat i kończy działanie preprocesora i kompilatora. Dyrektywa **#pragma** działa zgodnie ze specyfikacją danej realizacji kompilatora. Dyrektywa **#line** powoduje numerowanie linii kodu źródłowego.

# ROZDZIAŁ 14

## PLIKI

## 14.1. Wstęp

Z punktu widzenia programisty ważne jest, w jaki sposób język C obsługuje pliki. Najprostsza definicja pliku mówi, że plik (ang. file) jest ciągiem bajtów, z których każdy może być oddzielnie odczytany. Z punktu widzenia architektury komputera, plik jest wydzielonym obszarem pamięci posiadającym nazwę. Dla systemu operacyjnego plik jest dość skomplikowaną strukturą – np. plik może być przechowywany w kilku oddzielnych fragmentach.

Plik jest dostępny za pośrednictwem wskaźnika do struktury, który jest zdefiniowany w standardowym pliku nagłówkowym **<stdio.h>** jako FILE. Struktura zawiera składowe, które opisują aktualny stan pliku. Abstrakcyjnie możemy traktować plik jak ciąg znaków, które są przetwarzane sekwencyjnie. System otwiera trzy standardowe pliki (definiuje trzy wskaźniki plikowe).

| Nazwa pliku | Wskaźnik pliku | Podłączenie |
|---|---|---|
| Standardowe wejście | stdin | klawiatura |
| Standardowe wyjście | stdout | ekran |
| Standardowe wyjście do błędów | stderr | ekran |

Większość programów czyta i zapisuje dane w pamięci dyskowej. Dyskowe operacje wejścia/wyjścia są wykonywane na plikach. Program w języku C może odczytywać i zapisywać pliki na różne sposoby. Rozróżniamy standardowe wejście/wyjście (zwane także wysokiego poziomu lub strumieniowym) oraz systemowe wejście/wyjście (zwane także niskopoziomowym). Standardowe wejście/wyjście wysokiego poziomu (ang. *standard high-level I/O*) wykorzystuje funkcje biblioteczne i definicje znajdujące się w pliku **<stdio.h>.** Systemowe wejście/wyjście (ang. *low-level I/O*) korzysta z podstawowych usług udostępnianych przez system operacyjny.
Pakiet standardowego wejścia/wyjścia realizuje odczyt i zapis danych na cztery różne sposoby:

- znakowe (obsługa: **getche(), putchar()**)
- łańcuchowe (obsługa: **gets(), puts()**)
- sformatowane (obsługa: **printf(), scanf()**)
- rekordowe

Do transferu danych (obsługa dyskowego wejścia/wyjścia) używamy wyspecjalizowanych funkcji:

- wejście/wyjście znakowe (obsługa: **getc(), putc()**)
- wejście/wyjście łańcuchowe (obsługa: **fgets(), fputs()**)
- wejście/wyjście sformatowane (obsługa: **fprintf(), fscanf()**)
- wejście/wyjście rekordowe (obsługa: **fread(), fwrite()**)

Aby zrealizować operacje dyskowe systemowe wejście/wyjście dla obsługi rekordów stosuje funkcje **read()** i **write().**

Zgodnie ze standardem ANSI operacje plikowe wejścia/wyjścia mogą otwierać plik w **trybie tekstowym** lub **w trybie binarnym.** Istnienie tych dwóch trybów spowodowane jest próbą pogodzenia standardów używanych w różnych systemach operacyjnych (UNIX, WINDOWS, Macintosh).

Główne różnice to zapamiętywanie znaku nowej linii i oznaczenie końca pliku. Istnieje także różnica w zapisie liczb na dysku. W formacie tekstowym, liczby są zapisywane, jako łańcuchy znaków. W formacie binarnym liczby są zapisywane tak jak w pamięci: dwa bajty na liczbę całkowitą i cztery na liczbę zmiennoprzecinkową. Niektóre funkcje plikowe wejścia/wyjścia zapisują liczby jako tekst, inne zapisują je w formacie binarnym.

Standardowy pakiet wejścia/wyjścia ma dwie zalety nad systemowa obsługą wejścia/wyjścia. Po pierwsze zawiera on wiele wyspecjalizowanych funkcji i makrodefinicji do realizacji najróżniejszych zadań. Po drugie standardowe wejście/wyjście jest buforowane. Obsługa z buforem oznacza, że dane są przesyłane w dużych porcjach, zwykle po 512 bajtów na raz, a nie po jednym bajcie. Dzięki temu osiągamy znaczne szybkości przesyłania danych.

## 14.2. Znakowe wejście/wyjście

Bardzo często chcemy pobierać dane z klawiatury i zapisać wprowadzone informacje na dysku. Tego typu operacje można wykonać na wiele sposobów. Bardzo prosty program realizujący to zadanie jest pokazany na listingu 14.1.

Listing 14.1 Zapis znaków do pliku, znakowe we/wy

```
#include <stdio.h>
#include <conio.h>
#define PLIK "c:\\Prog_C\\test_1.txt"
int main()
{FILE *ws;
 char zn;
 ws = fopen (PLIK,"w");
 printf("Wprowadz tekst :\n");
 while ( (zn=getche()) != '\r')
    putc(zn,ws);
 fclose(ws);
 return 0;
}
```

Po uruchomieniu tego programu na ekranie pojawi się komunikat:

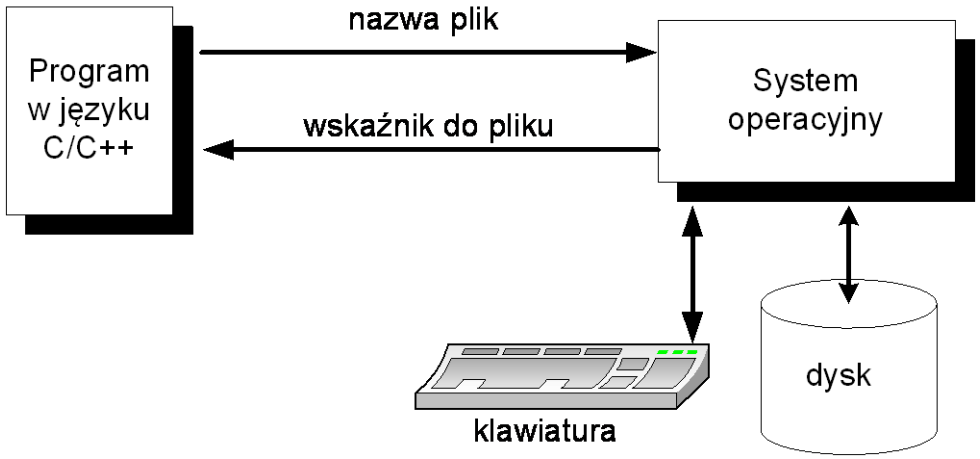
```
Wprowadz tekst :
```

Wpisujemy dowolny tekst, a następnie naciskamy klawisz ENTER. Wprowadzony z klawiatury tekst został zapisany na dysku **c:** w katalogu **Prog_C** w pliku o nazwie **test_1.txt.**

Aby zobaczyć ten plik możemy wykorzystać prosty edytor tekstu, np. NOTATNIK w systemie WINDOWS.

Ta dość prosta operacja, jaką jest zapisanie znaków w pliku na dysku, z punktu widzenia systemu jest skomplikowanym zadaniem. Formalnie chcemy stworzyć plik dyskowy i umieścić w nim nasze dane. Aby zrealizować zapis lub odczyt danych plikowych z dysku, plik musi być otwarty. Musi istnieć wymiana informacji pomiędzy programem i systemem operacyjnym.

Program dostarcza systemowi operacyjnemu nazwę pliku i jego położenie oraz inne potrzebne informacje (np. czy będziemy zapisywać w pliku, czy czytać z pliku). Potrzebne informacje przechowywane są w strukturze **FILE** zadeklarowanej w pliku nagłówkowych **<stdio.h>.** Należy zdawać sobie sprawę z przepływu informacji podczas realizacji standardowych operacji wejścia/wyjścia (rysunek 14.1).




Rys.14.1 Przepływ informacji podczas otwierania pliku

Struktura FILE przechowuje informacje o pliku takie jak np. aktualny rozmiar, czy położenie jego buforów danych. Nasz program na samym początku deklaruje zmienną wskaźnikową do FILE:

```
FILE *ws;
```

A następnie otwiera plik korzystając z funkcji **fopen(),** zadeklarowanej w pliku nagłówkowym <stdio.h>. Formalnie otwieramy plik o nazwie **test_1.txt**, który pamiętany jest na dysku **c:** i w katalogu **Prog_C** za pomocą instrukcji:

```
ws = fopen (”test_1.txt”,"w");
```

lub w specyfikując dodatkowo ścieżkę dostępu:

```
ws = fopen (“c:\\Prog_C\\test_1.txt“,"w");
```

W naszym przykładzie wykorzystaliśmy dyrektywę preprocesora:

```
#define PLIK "c:\\Prog_C\\test_1.txt"
.....................................
ws = fopen (PLIK,"w");
```

Funkcja **fopen()** wymaga argumentów. Pierwszym argumentem jest adres łańcucha zawierającego nazwę pliku, może też zawierać dodatkowo ścieżkę dostępu. Drugim argumentem jest specyfikator trybu otwarcia. W naszym przykładzie specyfikator ”**w**” informuje, że plik będzie otwarty do zapisu.
Pełny zestaw specyfikatorów trybów otwarcia podaje tabela 14.1.

Tabela 14.1 Specyfikatory trybów

| Tryb | Opis |
|---|---|
| ”r” | Otwiera plik tekstowy do czytania |
| ”w” | Otwiera plik tekstowy do zapisu. Jeżeli plik istniej, usuwa zawartość i umieszcza nowe dane. Jeżeli plik nie istnieje, zostaje utworzony. |
| ”a” | Otwiera plik do zapisu. Jeżeli plik istnieje, dopisuje nowe dane na końcu istniejących danych. Jeżeli plik nie istnieje zostaje utworzony. |
| ”r+” | Otwiera plik tekstowy do uaktualnienia, zezwala na zapis i czytanie |
| ”w+” | Otwiera plik tekstowy do uaktualnienia, zezwala na zapis i czytanie. Jeżeli plik istnieje, usuwa zawartość. Jeżeli plik nie istnieje jest tworzony. |
| ”a+” | Otwiera plik tekstowy do uaktualnienia, zezwala na zapis i czytanie. Jeżeli plik istnieje, dopisuje nowe dane na końcu. Jeżeli plik nie istnieje jest tworzony. Odczyt obejmuje cały plik, zapis polega na dodawaniu nowego tekstu. |
| ”rb”, ”wb”, ”ab”, ”rb+”, ”r+b”, ”wb+”, ”w+b”, ”ab+”, ”a+b” | Wymienione specyfikatory mają takie samo znaczenie jak powyższe, dotyczą **plików binarnych** |

Po pomyślnym otwarciu pliku można realizować zapisywanie tekstu:

```
while ( (zn=getche()) != '\r')
 putc(zn,ws);
```

Funkcja **getche()** odczytuje znaki z klawiatury dopóki realizowany jest warunek pętli **while** (dopóki użytkownik nie naciśnie klawisza ENTER). Zapis znaku w pliku wykonuje instrukcja:

```
putc(zn,ws);
```

Funkcja **putc()** jest podobna do funkcji **putch()** i **putchar().** Różni się tym, że zapisuje do pliku. Plik identyfikowany jest za pomocą struktury **FILE.** Struktura **FILE** jest wskazywana przez zmienną wskaźnikową **ws**, którą otrzymaliśmy po pomyślnym otwarciu pliku. Nie odwołujemy się do pliku przez jego nazwę, ale poprzez adres zapamiętany w zmiennej **ws**. Po zakończeniu zapisywania danych plik musi być zamknięty:

```
fclose(ws);
```

Informujemy system, że plik, którego adres umieszczony jest w zmiennej **ws** musi być zamknięty.
Gdy nasze dane zostały już zapisane w pliku o nazwie test_1.txt prawdopodobnie będziemy chcieli odczytać te informacje. Program czytający pojedyncze znaki z pliku o nazwie test_1.txt może mieć postać pokazana na listingu 14.2.

Listing 14.2. Odczyt znaków z pliku

```
/* znakowe we/wy, odczyt */
#include <stdio.h>
#include <conio.h>
#define PLIK "c:\\Prog_C\\test_1.txt"
int main()
{FILE *ws;
 char zn;
 ws = fopen (PLIK,"r");
 while ( (zn=getc(ws)) != EOF)
    printf("%c", zn);
 fclose(ws);
 getche();
 return 0;
}
```

Plik do odczytu musi być otwarty:

```
ws = fopen (PLIK,"r");
```

Czytanie znaków z pliku realizuje pętla **while**:

```
while ( (zn=getc(ws)) != EOF)
   printf("%c", zn);
```

Funkcja **getc(ws)** pobiera jeden znak z pliku identyfikowanego dzięki zmiennej wskaźnikowe **ws**, a funkcja **printf()** umieszcza znak na ekranie. Pętla **while** będzie działała tak długo, dopóki nie napotkany będzie symbol **EOF** – znak końca pliku. Sygnał końca pliku generowany jest przez system operacyjny. **EOF** jest to wartość całkowita wysyłana do programu przez system operacyjny i zdefiniowana w pliku nagłówkowym **<stdio.h>** jako wartość -1.

Korzystając z funkcji **fopen()** należy pamiętać, że jeżeli nie będziemy w stanie otworzyć pliku, to program nie będzie działał prawidłowo.

Dlatego ważne jest sprawdzenie czy plik został otwarty prawidłowo. Funkcja **fopen()** zwraca wartość, 0 (która jest zdefiniowana jako **NULL** w pliku nagłówkowym **<stdio.h>)** jeżeli plik nie może być otwarty. Następujące instrukcje mogą obsługiwać błędy otwarcia pliku:

```
if ( ws = fopen(PLIK,"r") ) == NULL)
    { printf("\n Nie moge otworzyc pliku");
      exit(1);
    }
```

Jeżeli funkcja **fopen()** zwraca **NULL**, wypisywane jest komunikat o błędzie i wykonywana jest instrukcja **exit(),** co powoduje natychmiastowe zakończenie programu. W programie pokazanym na listingu 14.3 następuje sprawdzenie, czy plik może być otwarty.

Listing 14.3. Sprawdzanie możliwości otwarcia pliku

```
/* znakowe we/wy, odczyt */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define PLIK "c:\\Prog_C\\test_1.txt"
int main()
{FILE *ws;
 char zn;
 if ( (ws = fopen(PLIK, "r")) == NULL )
      {
       printf("\n Nie mozna otworzyc pliku");
       getche();
       exit(1);
      }
 ws = fopen (PLIK,"r");
 while ( (zn=getc(ws)) != EOF)
    printf("%c", zn);
 fclose(ws);
 getche();
 return 0;
}
```

Należy zauważyć, że funkcja **fclose()** zwraca wartość **0**, jeśli operacja zamykania pliku się powiodła, w przeciwnym wypadku zwraca EOF :

```
if ( fclose(ws) != 0 )
    printf("\nBlad zamykania pliku");
```

Zamknięcie pliku może się nie udać, jeśli na dysku nie ma wolnego miejsca lub np. została wyjęta dyskietka z napędu.
Zapisywanie łańcuchów znaków do pliku i odczytywanie łańcuchów z pliku jest bardzo podobne do zapisywania i odczytywania znaków z pliku. W programie pokazanym na listingu 14.4 do zapisu łańcuchów wykorzystano funkcję **fputs().**

Po uruchomieniu programu za pomocą klawiatury należy wpisać serię łańcuchów. Ponieważ łańcuch jest zadeklarowany, jako tablica:

```
char napis[81];
```

należy pamiętać, aby nie wprowadzić więcej jak 80 znaków. Wykorzystano funkcję biblioteczną **strlen(),** która podaje długość łańcucha.

Listing 14.4. Zapis łańcuchów do pliku

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define PLIK "c:\\Prog_C\\test_2.txt"
int main()
{FILE *ws;
 char napis[81];
 ws = fopen (PLIK,"w");
   while (strlen ( gets(napis)) > 0 )
      {fputs(napis,ws);
       fputs("\n",ws);
      }
 fclose(ws);
 getche();
 return 0;
}
```

Wprowadzanie łańcuchów kontroluje pętla **while**:

```
while (strlen ( gets(napis)) > 0 )
{ fputs(napis,ws);
  fputs("\n",ws);
}
```

Aby zakończyć wprowadzanie tekstu, należy wcisnąć klawisz Enter na początku wiersza (wtedy funkcja **strlen()** zwróci wartość 0). Funkcja **fputs()** nie dodaje automatycznie znaku nowej linii, dlatego musimy sami zadbać o wprowadzenie tego znacznika, aby potem można było łatwiej wczytywać łańcuchy z pliku.
Do odczytywania łańcuchów z pliku wygodnie jest użyć funkcji **fgets(),** tak jak w programie pokazanym na kolejnym listingu 14.5.
Funkcja **fgets()** wymaga trzech parametrów: pierwszym jest adres gdzie ma być zapisany łańcuch (w naszym przypadku jest to nazwa tablicy), drugim parametrem jest maksymalna długość łańcucha, trzecim parametrem jest wskaźnik do struktury FILE związany z plikiem. Wczytywanie kolejnych łańcuchów z pliku kontroluje pętla while:

```
while (fgets ( napis, 80, ws) != NULL )
 printf("%s", napis);
```

Listing 14.5. Odczyt łańcuchów z pliku

```
/* napisowe we/wy, odczyt */
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define PLIK "c:\\Prog_C\\test_2.txt"
int main()
{FILE *ws;
 char napis[81];
 ws = fopen (PLIK,"r");
 while (fgets ( napis, 80, ws) != NULL )
      printf("%s", napis);
 fclose(ws);
 getche();
 return 0;
}
```

Omawialiśmy dotychczas metody zapisywania znaków i łańcuchów do pliku. Zapisywanie liczb do pliku realizuje się bardzo podobnie. Należy najpierw otworzyć plik do zapisu, wykorzystać odpowiednią funkcje obsługującą zapis liczb do pliku a po zakończeniu zapisywania należy plik zamknąć.
W pokazanym na listingu 14.6 programie, wprowadzamy z klawiatury dane liczbowe (mogą to być np. temperatury zanotowane w kolejnych dniach). Temperatury będziemy przechowywać w tablicy.
Obsługę wprowadzania temperatur z klawiatury realizuje następujący fragment:

```
printf("\n wprowadz temeratury, 0 aby konczyc \n");
do
 {printf("temperatura w dniu nr %d = ",num);
  scanf("%f",&temperatura[num]);
 } while (temperatura[num++] > 0);
```

W naszym przykładzie wprowadziliśmy z klawiatury następujące temperatury: 10.1, 20.2, 30.3 i 40.4. Aby zakończyć wprowadzanie danych zostało wprowadzone na końcu zero (0). Funkcja **printf()** spowoduje wyświetlenie wprowadzonych danych na ekranie monitora. Aby dane mogły być zapisane w pliku dyskowym, musimy wprowadzić odpowiednie instrukcje i dyrektywy w programie. Po pierwsze musimy otworzyć plik, ponieważ jest to niezbędne do zapewnienia komunikacji pomiędzy programem a systemem operacyjnym. Następnie musimy ustalić obszary komunikacji pomiędzy plikiem i naszym programem otworzyć plik. Do tego potrzebna jest nam struktura, która przechowuje informacje o pliku.
W dyrektywie preprocesora:

```
#define PLIK "c:\\Prog_C\\test_3.txt"
```

umieszczamy nazwę naszego pliku (oczywiście nazwa pliku może być wprowadzona także z klawiatury).

Listing 14.6. zapis liczb do pliku

```
//zapisuje formatowane dane do pliku, fprintf()
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define PLIK "c:\\Prog_C\\test_3.txt"
int main()
{FILE *ws;
 float temperatura[20];
 int num = 0;
 ws = fopen (PLIK,"w");
 printf("\n wprowadz temeratury, 0 aby konczyc \n");
 do
  {printf("temperatura w dniu nr %d = ",num);
   scanf("%f",&temperatura[num]);
  } while (temperatura[num++] > 0);
 for (int i = 0; i < num; i++)
  printf("\n dzien %d T = %6.1f",i,temperatura[i]);
 fprintf(ws,"%d",num);
 for (int j = 0; j < num; j++)
    fprintf(ws,"\n%d %f",j,temperatura[j]);
 fclose(ws);
 getche();
 return 0;
}
```

W programie deklarujemy zmienna typu wskaźnik do FILE za pomocą instrukcji:

```
FILE *ws;
```

a następnie otwieramy plik:

```
ws = fopen (PLIK,"w");
```

W tym miejscu przekazana jest informacja, że powinien być otwarty plik o nazwie **test_3.txt** w katalogu **Prog_C**. W instrukcji umieszczono specyfikator, „w”, który informuje, że będziemy pisać do pliku. Funkcja **fopen()** zwraca wskaźnik do struktury FILE związanej z naszym plikiem, którą zapamiętujemy w zmiennej **ws.**

Łańcuch ”w” umieszczony w funkcji **fopen()** określa typ otwartego pliku. W języku C mamy następujące specyfikatory plików:

| | |
|---|---|
| ”r” | otwórz do odczytu. Plik musi istnieć |
| ”w” | otwórz do zapisu. Jeśli plik istnieje, jego zawartość zostanie utracona. Jeśli nie istnieje, zostanie utworzony. |
| ”a” | otwórz do dopisywania. Dane będą dopisywane na koniec istniejącego pliku, albo nowy plik zostanie utworzony. |
| ”r+” | otwórz zarówno do odczytu jak i do zapisu. Plik musi istnieć |

”w+” otwórz zarówno do odczytu jak i do zapisu. Jeśli plik istnieje, jego zawartość zostanie utracona

”a+” otwórz zarówno do odczytu jak i do dopisywania. Jeśli plik nie istnieje, to zostanie utworzony

Po otwarci pliku możemy zacząć pisać do niego. W naszym przykładzie zasadniczą role odgrywa funkcja **fprintf().** Za pomocą tej funkcji zapisujemy wartości dwóch zmiennych do pliku. Jak widzimy, konstrukcja tej funkcji jest bardzo podobna do konstrukcji funkcji **printf(),** z wyjątkiem tego, że jako pierwszy argument występuje wskaźnik do FILE. Pozostałe argumenty są takie same jak w funkcji **printf().** Zapisywanie danych do pliku odbywa się pod kontrolą pętli:

```
fprintf(ws,"%d",num);
for (int j = 0; j < num; j++)
     fprintf(ws,"\n%d %f",j,temperatura[j]);
```

Wprowadzone dane zostaną zapisane w pliku. Po zakończeniu zapisywania do pliku, musimy go zamknąć. Zamykanie pliku wykonywane jest dzięki instrukcji:

```
fclose(ws);
```

Możemy sprawdzić działanie naszego programu. W systemie Windows za pomocą np. Notatnika możemy przeczytać zawartość utworzonego pliku. W naszym przypadku zawartość pliku jest następująca:

```
5
0    10.1
1    20.2
2    30.3
3    40.4
4     0.0
```

Wprowadzone z klawiatury liczby zostały zapisane w pliku. Na kolejnym listingu 14.7 pokazany jest program, dzięki któremu odczytujemy liczby zapisane w pliku. Struktura programu odczytującego dane z pliku jest taka sama jak programu służącego do zapisywania danych w pliku. Po otwarciu pliku za pomocą instrukcji:

```
ws = fopen (PLIK,"r");
```

następuje wczytywanie danych z pliku realizowanych za pomocą funkcji **fscanf():**

```
fscanf(ws, "%d",&num);
```

Funkcja **fscanf()** jest podobna do **scanf(),** z tym, że pierwszym parametrem jest wskaźnik do FILE. Pokazana instrukcja odczytuje ilość wprowadzonych temperatur.

Listing 14.7. czyta liczby z pliku, fscanf()

```
#include <stdio.h>
#include <conio.h>
#define PLIK "c:\\Prog_C\\test_3.txt"

int main()
{FILE *ws;
 float temperatura[20];
 int num=0;
 ws = fopen (PLIK,"r");
 fscanf(ws, "%d",&num);
 for (int i = 0; i<num; i++)
   { fscanf(ws, "%d %f",&i, &temperatura[i]) ;
     printf("\n dzien %d T = %f", i,temperatura[i]);
   }
 fclose(ws);
 getche();
 return 0;
}
```

Wczytywanie temperatur w kolejnych dniach realizuje pętla:

```
for (int i = 0; i<num; i++)
   { fscanf(ws, "%d %f",&i, &temperatura[i]) ;
     printf("\n dzien %d T = %f", i,temperatura[i]);
   }
```

Za pomocą pokazanego schematu można zapisywać do pliku i odczytywać z pliku dowolne kombinacje znaków, łańcuchów, liczb całkowitych i liczb zmiennoprzecinkowych.
Możemy oczywiście zapisać dowolna ilość tekstu w pliku. W kolejnym przykładzie widzimy program, dzięki któremu możemy zapisać bajkę La Fontaine'a w pliku. Wpisywanie łańcuchów kontroluje pętla:

```
while ( strlen( gets(napis) ) > 0 )
   {
     fputs (napis, ws);
     fputs("\n", ws);
   }
```

Tekst wpisujemy wierszami, każdorazowo wiersz kończymy naciskając klawisz Enter, kursor przechodzi do następnej linii i wpisywanie tekstu zaczynamy od nowa. Aby zakończyć cały program trzeba nacisnąć klawisz Enter na początku wiersza. Ponieważ funkcja **fputs()** nie daje automatycznie znaku nowej linii, musimy go dodać w programie za pomocą instrukcji:

```
fputs("\n", ws);
```

aby można było później łatwiej czytać tekst z pliku.

Listing 14.8. Zapisuje napisy do pliku

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{FILE *ws;
 char napis[81];
 printf("\n Wprowadz tekst, enter, enter konczy
          zapis\n");
 ws = fopen("c:\\Prog_C\\test_4.txt", "w");
 while ( strlen( gets(napis) ) > 0 )
     {fputs (napis, ws);
      fputs("\n", ws);
     }
 fclose(ws);
 return 0;
}
```

Po uruchomieniu programu wpisujemy następujący tekst (pamiętamy, że nie wprowadzamy polskich znaków takich jak np. ‘ ą ‘ czy ‘ ł ‘):

*Jean de La Fontaine*
*Lis i winogrona*
*Lis pewien, łgarz i filut, wychudły, zgłodniały,*
*Zobaczył winogrona rosnące wysoko,*
*Owoc przejrzysty okryty powłoką,*
*Zdał się lisowi dojrzały.*
*Wiec rad z uczty, wytężył swoją chudą postać,*
*Skoczył, sięgnął, lecz nie mógł do jagód się dostać.*
*Wprędce przeto zaniechał daremnych podskoków*
*I rzekł: "Kwaśne, zielone, dobre dla żarłoków."*

Ten wiersz zostaje zapisany w pliku o nazwie **test_4.txt** na dysku C. Krótki program pokazany na listingu 14.9 służący do odczytu ma postać:

Listing 14.9. Czyta napisy z pliku

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{FILE *ws;
 char napis[81];
 ws = fopen("c:\\Prog_C\\test_4.txt", "r");
 while ( fgets( napis, 80, ws) != NULL )
     printf("%s", napis);
 fclose(ws);
 return 0;
}
```

Po uruchomieniu tego programu mamy następujący wydruk na ekranie:

```
Jean de La Fontaine
Lis i winogrona
Lis pewien, lgarz i filut, wychudly, zglodnialy,
Zobaczyl winogrona rosnace wysoko,
Owoc przejrzysty okryty powloka,
Zdal sie lisowi dojrzaly.
Wiec rad z uczty, wytezyl swoja chuda postac,
Skoczyl, siegnal, lecz nie mogl do jagod sie dostac.
Wpredce przeto zaniechal daremnych podskokow
I rzekl: "Kwasne, zielone, dobre dla zarlokow."
```

Możemy zastosować tryb binarny do odczytania danych z pliku. Pokazany poniżej na listingu 14.10 program przegląda plik bajt po bajcie w trybie binarnym i wyświetla zawartość pliku (wynikowy zrzut ekranu pokazany jest na rys. 14.2). W naszym przykładzie odczytamy w ten sposób plik o nazwie **tekst_4.txt**, w którym zapisaliśmy poprzednio bajkę La Fonatine'a. Na wydruku otrzymamy kod każdego zapisanego znaku w trybie szesnastkowym oraz sam znak, (jeżeli jest możliwy do wyświetlenia.

Plik od odczytu otwierany jest w instrukcjach:

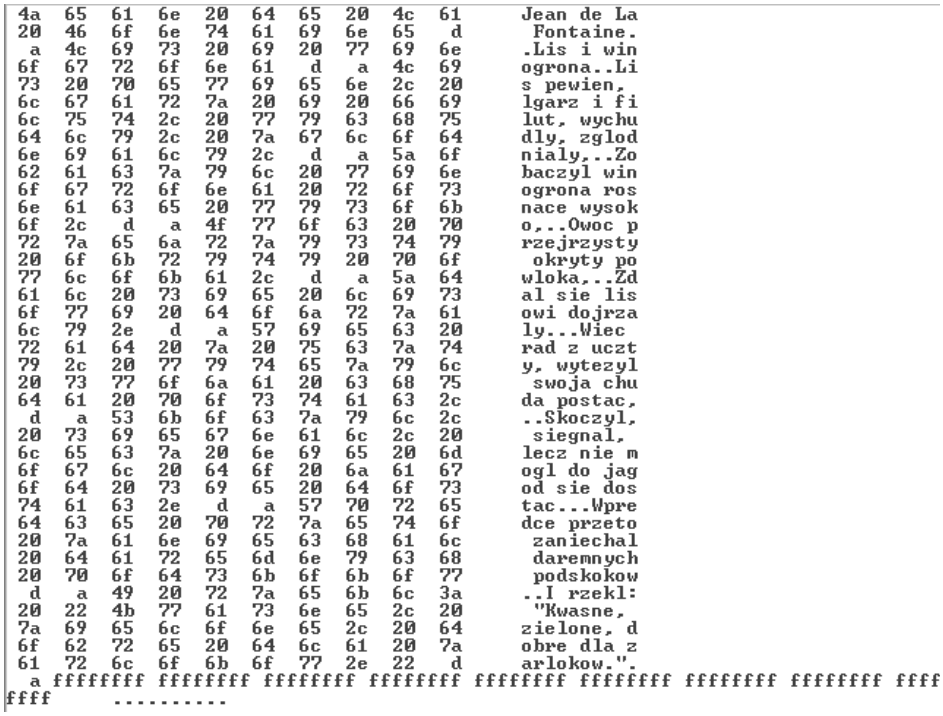
```
if ((ws = fopen("c:\\Prog_C\\test_4.txt","rb"))== NULL)
   {
      printf("\n nie mozna otworzyc pliku");
      exit(1);
   }
```

Zwróćmy uwagę, że specyfikator "rb" wymusza traktowanie otwieranego pliku, jako pliku binarnego. W instrukcji **if** sprawdzamy także, czy plik otworzył się poprawnie. Możemy sprawdzić, że kody zapisane z lewej strony są kodami znaków zapisanych z prawej strony. Z prawej strony wydruku widzimy na początku zapis "Jean". Łatwo możemy sprawdzić, że 4a to szesnastkowo zapisana liczba dziesiętna 74, a to odpowiada kodowi ASCII znaku **J.** Podobnie liczba 65 to kod znaku **e**, 61 to znak **a**, 6e to znak **n**. Kod 20 oznacza niedrukowalny znak spacji. Tekst zapisany w pliku kończy się kropką (kod 2e) oraz znakiem cudzysłowu – kod 22. Na końcu pliku program zaczyna czytać znaki końca pliku, które są zapisywane, jako ffff szesnastkowo. W programie sprawdzamy, czy przeczytano już ostatni znak w pliku, sprawdzając, czy napotkano sygnał końca pliku (EOF) generowany przez system operacyjny. Należy pamiętać, że EOF nie jest znakiem. W rzeczywistości jest to liczba całkowita wysyłana do programu przez system operacyjny i zdefiniowana w pliku <stdio.h> jako wartość -1. Żaden znak o tej wartości nie jest przechowywany w pliku na dysku.

Listing 14.10. Czyta napisy z pliku w trybie binarnym

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define LEN 10
#define TRUE 1
#define FALSE 0
int main()
{FILE *ws;
 int zn;
 int not_eof;
 unsigned char napis[LEN+1];
 if((ws=fopen("c:\\Prog_C\\test_4.txt","rb"))== NULL)
        { printf("\n nie mozna otworzyc pliku");
          exit(1);
        }
 not_eof = TRUE;
 do
   {for (int j = 0; j < LEN; j++)
       {if ( (zn = getc(ws)) == EOF)
           not_eof = FALSE;
        printf(("%3x ", zn);
        if (zn > 31)
           *(napis+j) = zn;
        else
           *(napis+j) = '.';
       }
    *(napis+j) = '\0';
    printf("    %s\n", napis);
   } while (not_eof == TRUE);
   fclose(ws);
  getche();
  return 0;
}
```

Stosując tryb tekstowy i tryb binarny należy pamiętać o wieloznaczności tych terminów. Standard ANSI C udostępnia dwa tryby otwierania plików: binarny i tekstowy. Wiele systemów operacyjnych posiada dwa formaty plików: binarny i tekstowy. Informacje mogą być przechowywane, jako dane binarne lub tekstowe. Wszystkie te znaczenia są powiązane, ale nie tożsame. Plik w formacie tekstowym może zostać otwarty w trybie binarnym ( tak jak w naszym przykładzie). W pliku binarnym można zapisać tekst. Rozsądnie jest jednak dane binarne zapisywać w pliku binarnym korzystając z trybu binarnego.




Rys. 14.2 Kopia wydruku ekranowego z programu 14.10

Podobnie pliki tekstowe powinno się otwierać w trybie tekstowym po to, aby pobrać z nich dane tekstowe.

W poprzednich programach użyliśmy funkcji formatowanego wejścia/wyjścia **fscanf()** i **fprintf()** do odczytu i zapisu liczb w pliku. Użycie tych funkcji nie jest ekonomiczne, ponieważ każda cyfra jest przechowywana, jako jeden znak, co powoduje, że dane tak zapisane zajmują dużo miejsca na dysku. Język C dostarcza bardziej efektywnych narzędzi do optymalnego zapisu danych na dysku. Tym uniwersalnym narzędziem jest rekordowe wejście/wyjście zwane czasami blokowym wejściem/wyjściem. Rekordowe wejście/wyjście zapisuje liczby w formacie binarnym w takiej postaci, w jakiej są one przechowywane w pamięci operacyjnej. Rekordowe wejście/wyjście umożliwia zapisywanie zarówno pojedynczych liczb jak i tablic, struktur, tablic struktur i innych typów danych w pojedynczej instrukcji. W standardowym pakiecie wejścia/wyjścia wymiana danych w formie binarnej zajmują się funkcje **fread()** i **fwrite().**

Prototypy funkcji **fread()** i **fwrite()** mają postać:

```
size_t fread(void *ptr,size_t size,size_t n,FILE *stream);
size_t fwrite(const void *ptr,size_t size,size_t n,FILE
                *stream);
```

W prototypie funkcji **fread()** argument **ptr** jest wskaźnikiem do obszaru pamięci, w którym mają zostać zapisane dane odczytane z pliku.

W prototypie funkcji **fwrite()**, argument **ptr** wskazuje na obszar z zapisywanymi informacjami. Drugi parametr **size** jest długością jednostkowej pozycji danych. Trzeci parametr **n** określa liczbę pozycji danych. Ostatni argument **stream** jest wskaźnikiem do otwartego wcześniej pliku. Funkcja **fread()** zwraca liczbę przeczytanych pozycji, funkcja **fwrite()** zwraca liczbę zapisanych pozycji.
W przykładzie 14.11 zademonstrujemy użycie funkcji **fwrite()** do zapisu tablicy w pliku dyskowym oraz funkcji **fread()** do odczytu tej tablicy.

Listing 14.11. Zapis tablicy do pliku w trybie binarnym, fwrite()

```
#include <stdio.h>
#include <conio.h>
int tab[10] = {1,2,3,4,5} ;

int main()
{ FILE *ws;
  ws = fopen("c:\\Prog_C\\test_5.rec", "wb");
  fwrite(tab, sizeof(tab), 1, ws);
  puts("zapisano");
  getche();
  return 0;
}
```

W programie tablica dziesięcioelementowa **tab[10]** została zainicjalizowana 5 wartościami. W instrukcji:

```
ws = fopen("c:\\Prog_C\\test_5.rec", "wb");
```

otworzono do zapisu plik o nazwie **test_5.rec**. Plik jest otworzony do zapisu w trybie binarnym. Instrukcja:

```
fwrite(tab, sizeof(tab), 1, ws);
```

zapisuje tablicę do pliku. W naszym przykładzie argument **tab** jest adresem tablicy, która ma zostać zapisana. Drugim argumentem jest rozmiar tablicy. Trzecim argumentem jest liczba takich tablic, które chcemy zapisać za jednym razem. W naszym przypadku mamy jedną tablicę. Ostatni argument **ws** jest wskaźnikiem do pliku, do którego chcemy zapisać dane. Aby odczytać dane zapisane na pliku za pomocą pokazanego programu, możemy się posłużyć kolejnym programem, pokazanym na listingu 14.12, wykorzystującym funkcję **fread().** W instrukcjach:

```
if ((ws=fopen("c:\\Prog_C\\test_5.rec","rb")) == NULL)
   {  printf ("\n nie mozna otworzyc pliku");
      getche();
      exit(1);
   }
```

otwieramy plik do odczytu i sprawdzamy, czy otworzył się on prawidłowo.

Listing 14.12. Odczyt tablicy z pliku binarnego, fread()

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int tab[10];
int main()
{FILE *ws;
 if ((ws=fopen("c:\\Prog_C\\test_5.rec","rb"))== NULL)
   {printf ("\n nie mozna otworzyc pliku");
     getche();
     exit(1);
    }
 fread(tab, sizeof(tab),1,ws);        //odczyt tablicy
 for (int i = 0; i < 10; i++)         //wydruk tablicy
           printf(" %d",tab[i]);
 fclose(ws);
  getche();
  return 0;
}
```

W kolejnej instrukcji:

```
fread(tab, sizeof(tab), 1, ws);       //odczyt tablicy
```

dane są odczytywane z dysku i umieszczane w tablicy **tab[].** Aby wyświetlić odczytane elementy tablicy posługujemy się klasyczną postacią funkcji **printf():**

```
for (int i = 0; i < 10; i++)          //wydruk tablicy
     printf(" %d",tab[i]);
```

Odczytywanie i zapisywanie skomplikowanych danych za pomocą funkcji **fread()** i **fwrite()** działa bardziej efektywnie niż w przypadku wielokrotnego wywoływania funkcji **fprintf()** i **fscanf().**

Wyraźne korzyści osiągamy stosując funkcję **fwrite()** do zapisu danych przechowywanych w strukturach. W kolejnym przykładzie (program pokazany na listingu 14.13) wykorzystamy zapis danych do pliku w formacie binarnym dla przechowywania informacji osobowych. W realnym świecie instytucje przetwarzają ogromne ilości danych osobowych. W szpitalach np. bardzo często tworzone są dane o pacjentach. W naszym przykładzie utworzymy odpowiednią strukturę, w której umieszczone będą dane o pacjentach a następnie zapiszemy te dane w pliku. W tym programie wprowadzane są dane o pacjentach. Na początku utworzona zostaje struktura postaci:

```
struct pacjent
    {
      char imie[80];
      int nr_badania;
     double koszt;
    };
```

a następnie tworzymy zmienną strukturalną pt:

```
struct pacjent pt;
```

Listing 14.13. Zapis struktury do pliku binarnego, fwrite()

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct pacjent
    { char imie[80];
      int nr_badania;
      double koszt;
    };
int main()
{struct pacjent pt;
 char dane[81];
 FILE *ws;
 ws = fopen("c:\\Prog_C\\test_6.rec", "wb");
 do
   { printf("\nPodaj nazwisko: ");
     gets(pt.imie);
     printf("Poadj numer badania: ");
     gets(dane);
     pt.nr_badania = atoi(dane);
     printf("Poadaj koszt: ");
     gets(dane);
     pt.koszt = atof(dane);
     fwrite(&pt, sizeof(pt), 1, ws);
     printf("Czy wpisac kolejnego pacjenta (t/n)?: ");
   } while (getche() == 't');
 fclose(ws);
 getche();
 return 0;
}
```

W pętli

```
do
   { printf("\nPodaj nazwisko: ");
     gets(pt.imie);
     printf("Poadj numer badania: ");
     gets(dane);
     pt.nr_badania = atoi(dane);
     printf("Poadaj koszt: ");
     gets(dane);
     pt.koszt = atof(dane);
     fwrite(&pt, sizeof(pt), 1, ws);
     printf("Czy wpisac kolejnego pacjenta (t/n)?: ");
   } while (getche() == 't');
```

wprowadzamy z klawiatury potrzebne dane. Należy zwrócić uwagę, że wszystkie dane wprowadzamy jednolicie, jako dane znakowe. W miarę potrzeby stosujemy funkcje **atoi()** i **atof()**, aby dokonać konwersji ciągu znaków na liczby całkowite lub liczby zmiennoprzecinkowe. Dzięki warunkowi:

```
while (getche() == 't');
```

możemy wprowadzić dane dotyczące dowolnej ilości pacjentów. Zapis struktury do pliku realizuje instrukcja:

```
fwrite(&pt, sizeof(pt), 1, ws);
```

Pierwszym argumentem funkcji **fwrite()** jest adres struktury, która ma zostać zapisana. Drugim argumentem jest rozmiar struktury. Trzecim argumentem jest liczba struktur, które chcemy zapisać za jednym razem, ostatnim argumentem jest wskaźnik do otwartego pliku. Po uruchomieniu programu z klawiatury wprowadzane są kolejne dane. W wyniku działania programu zostanie utworzony plik o nazwie **test_6.rec** zapisany na dysku C. Aby odczytać dane musimy dysponować odpowiednim programem (pokazanym na listingu 14.14).

Listing 14.14. Odczyt struktury z pliku binarnego, fread()

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct pacjent
    { char imie[80];
      int nr_badania;
      double koszt;
    };

int main()
{struct pacjent pt;
 char dane[81];
 FILE *ws;
 if ((ws=fopen("c:\\Prog_C\\test_6.rec","rb"))==NULL)
      {printf("Nie mozna otworzyc pliku");
       getche();
       exit(1);
      }
 while ( fread (&pt, sizeof(pt), 1, ws) == 1 )
      {printf("\nNazwisko: %s\n", pt.imie);
       printf("\nnumer badania: %d\n", pt.nr_badania);
       printf("\nkoszt: %lf\n", pt.koszt);
      }
 fclose(ws);
 getche();
 return 0;
}
```

W pokazanym programie sprawdzamy poprawność otwarcia pliku:

```
if ((ws=fopen("c:\\Prog_C\\test_6.rec","rb"))==NULL)
       {printf("Nie mozna otworzyc pliku");
        getche();
        exit(1);
       }
```

Do odczytania danych z pliku wykorzystujemy funkcję **fread():**

```
fread (&pt, sizeof(pt), 1, ws )
```

Pierwszym parametrem tej funkcji jest adres struktury, w której maja być umieszczone dane, drugim jest rozmiar struktury. Ostatnim parametrem jest wskaźnik do otwartego do czytania pliku.
Funkcja **fread()** umieszcza odczytane dane w strukturze **pt**. Aby te dane wyświetlić korzystamy z funkcji **printf(),** a do poszczególnych elementów struktury odwołujemy się za pomocą operatora kropki:

```
while ( fread (&pt, sizeof(pt), 1, ws) == 1 )
{
    printf("\nNazwisko: %s\n", pt.imie);
    printf("\nnumer badania: %d\n", pt.nr_badania);
    printf("\nkoszt: %lf\n", pt.koszt);
}
```

Ponieważ dane były zapisane w formacie binarnym, musimy pamiętać, aby plik otworzy c także w trybie binarnym (specyfikatory ”wb” i ”rb” ).

# ROZDZIAŁ 15

## FUNKCJE BIBLIOTECZNE JĘZYKA C

## 15.1. Wstęp

Wiele użytecznych funkcji zostało zgromadzonych w plikach nagłówkowych, które tworzą ***bibliotekę języka C***. Plik nagłówkowy zawiera prototypy funkcji, stałe symboliczne i definicje funkcji bibliotecznych (nagłówkowych). Każda biblioteka funkcji języka C oprócz plików zatwierdzonych przez standard ANSI C zawiera własne dodatkowe pliki z niestandardowymi funkcjami. Należy przejrzeć dokumentacje techniczną danej implementacji języka C, aby zapoznać się z dodatkowymi możliwościami.

Poniżej przedstawiamy zestawienie (wybrane arbitralnie) plików nagłówkowych wykorzystywanych w implementacji języka C.

| Nr | Nazwa | Opis |
|---|---|---|
| 1 | alloc.h | Funkcje do zarządzania pamięcią |
| 2 | assert.h | Makra do testowania programów |
| 3 | bcd.h | Funkcje do operacji na liczbach binarnych |
| 4 | bios.h | Funkcje obsługujące procedury BIOS |
| 5 | complex.h | Funkcje do wykonywania operacji na liczbach zespolonych |
| 6 | conio.h | Funkcje do obsługi wejścia/wyjścia (konsola i DOS) |
| 7 | ctype.h | Makra do obsługi znaków |
| 8 | dir.h | Struktury, makra i funkcje do obsługi katalogów i ścieżek dostępu |
| 9 | dos.h | Różne stałe i funkcje potrzebne do obsługi DOS |
| 10 | errno.h | Stałe do obsługi błędów |
| 11 | fcntl.h | Stałe symboliczne do obsługi procedury open |
| 12 | float.h | Parametry procedur do obsługi liczb zmiennoprzecinkowych |
| 13 | generic.h | Makra do deklaracji klas |
| 14 | graphics.h | Prototypy funkcji graficznych |
| 15 | io.h | Struktury i deklaracje dla procedur wejście/wyjście |
| 16 | iomanip.h | Deklaracje i makra do obsługi strumieni wejście/wyjście |
| 17 | iostream.h | Deklaracje procedur strumieniowych wejścia/wyjścia |
| 18 | limits.h | Parametry środowiskowe i wartości stałych całkowitych |
| 19 | locale.h | Funkcje do obsługi informacji o języku, parametry lokalne |
| 20 | math.h | Prototypy funkcji matematycznych i makr |
| 21 | mem.h | Deklaracje funkcji do manipulowania pamięcią |
| 22 | process.h | Struktury i deklaracje do obsługi procesów |
| 23 | setjmp.h | Deklaracje i funkcje do obsługi dalekich skoków |
| 24 | share.h | Parametry potrzebne funkcjom obsługującym pliki |
| 25 | signal.h | Stałe i deklaracje do obsługi sygnałów |
| 26 | stdarg.h | Makra do obsługi listy argumentów w funkcjach |
| 27 | stdio.h | Typy i makra do obsługi standartowego wejścia/wyjścia |

| | | |
|---|---|---|
| 28 | stdiostr.h | Klasy C++ do obsługi struktury FILE |
| 29 | stdlib.h | Biblioteka standardowych procedur |
| 30 | string.h | Deklaracje procedur do obsługi łańcuchów |
| 31 | sys\stat.h | Stałe symboliczne do tworzenia plików |
| 32 | sys\timeb.h | Deklaracje funkcji ftime i struktury timeb |
| 33 | sys\types.h | Deklaracja typu time_t |
| 34 | time.h | Procedury do obsługi czasu |
| 35 | values.h | Definicje ważnych stałych (matematycznych i systemowych) |

Jest wiele argumentów przemawiających za tym, aby korzystać z funkcji bibliotecznych, dużą korzyścią jest oszczędność czasu.

## 15.2. Algorytm szybkiego sortowania – funkcja qsort()

W pliku nagłówkowym **stdlib.h** umieszczona jest funkcja **qsort(),** która realizuje algorytm szybkiego sortowania (*quick sort*). Wykorzystamy tą funkcje do posortowania zbioru imion.

Listing 15.1 Funkcja biblioteczna qsort()

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#define ILE 50          //maksymalna liczba imion
#define LEN 20          //maksymalna dlugosc imienia
char lista[ILE][LEN];
int por( const void *a, const void *b);
int main(void)
{  int  x, n;
   printf("\nPodaj ilosc imion do sortowania :");
   scanf("%d",&n);
   printf("\nPodaj imiona: \n\n");
   for (int i = 0; i<n; i++)
     scanf("%s", lista[i]);
   qsort((void *)lista, n, sizeof(lista[0]), por);
   printf("\n Lista posortowana :\n");
   for (x = 0; x < n; x++)
     printf("%s\n", lista[x]);
   getche();
   return 0;
}
int por( const void *a, const void *b)
{   return( strcmp((char *)a,(char *)b) );
}
```

W programie tworzona jest lista imion:

```
#define ILE 50          //maksymalna liczba imion
#define LEN 20          //maksymalna dlugosc imienia
char lista[ILE][LEN];
printf("\nPodaj ilosc imion do sortowania :");
scanf("%d",&n);
printf("\nPodaj imiona: \n\n");
for (int i = 0; i<n; i++)
   scanf("%s", lista[i]);
```

Formalnie mamy do czynienia z tablicą dwuwymiarową **lista[ ][ ],** która zarazem jest tablicą łańcuchów. Kolejność indeksów w tablicy jest ważna. Pierwszy indeks **ILE** określa liczbę napisów (tutaj imion), natomiast drugi indeks **LEN** określa długość łańcucha.
Oto przykład działania programu:

```
Podaj ilosc imion do sortowania :  4
Podaj imiona :
  Wacek
  Ala
  Ola
  Zenek
Lista posortowana :
  Ala
  Ola
  Wacek
  Zenek
```

Sortowanie imion (alfabetyczne) zostało wykonane za pomocą funkcji **qsort().** Ta funkcja porządkuje tablicę dowolnych obiektów danych. Prototyp tej funkcji ma postać:

```
void qsort (void *base, size_t nmemb, size_t size,
       int (*compar) (consta void *, consta void *));
```

Pierwszym argumentem funkcji jest wskaźnik do tablicy, która ma być sortowana, drugim argumentem jest liczba sortowanych elementów. Trzeci argument podaje rozmiar elementu tablicy, (jeżeli sortujemy np. tablicę typu **double** to tym argumentem jest **sizeof(double)).** Ostatnim parametrem jest wskaźnik do funkcji porządkującej. Funkcja porządkująca przyjmuje dwa argumenty, są nimi dwa wskaźniki do porównywanych pozycji. Funkcja zwraca dodatnia liczbę całkowitą, gdy pierwsza pozycja powinna znajdować się po drugiej pozycji, zwraca zero, gdy pozycje są identyczne i zwraca ujemną liczbę całkowitą, gdy druga pozycja powinna znajdować się po pozycji pierwszej.
Sortowanie i wydruk listy uporządkowanej realizuje fragment programu:

```
qsort((void *)lista, n, sizeof(lista[0]), por);
printf("\n Lista posortowana :\n");
for (x = 0; x < n; x++)
    printf("%s\n", lista[x]);
```

Funkcja biblioteczna **strcmp(string1, string2)** porównuje dwa łańcuchy i zwraca wartość całkowitą będącą wynikiem porównania:

| zwrócona wartość | znaczenie |
|---|---|
| mniejsze od zera | string1 jest mniejsze niż string2 |
| zero | string1 jest identyczny z string2 |
| większe od zera | string1 jest większy niż string2 |

W tym kontekście relacje oznaczają uporządkowanie w kolejności alfabetycznej. Jeden łańcuch jest mniejszy od drugiego, gdy występuje wcześniej w kolejności alfabetycznej. Tą właśnie funkcję wykorzystano do implementacji funkcji **por():**

```
int por( const void *a, const void *b)
{
  return( strcmp((char *)a,(char *)b) );
}
```

Funkcja **qsort()** jest bardzo uniwersalna – pozwala na sortowanie liczb całkowitych, zmiennoprzecinkowych, struktur, łańcuchów i innych. Jest to możliwe, ponieważ jednym z argumentów funkcji **qsort()** jest wskaźnik do funkcji porównującej. Tworząc odpowiednie funkcje porównujące możemy sortować w zasadzie dowolny typ zmiennych. Format funkcji porównującej jest określona w prototypie funkcji **qsort()** w następujący sposób:

```
int ( *compar) (const void *, const void *)
```

Parametrem funkcji **qsort()** jest wskaźnik do funkcji porównującej (dostarcza ją użytkownik), zwracana wartość jest typu **int**, funkcja porównująca wymaga dwóch argumentów, każdy z nich jest typu **const void.** Jest to bardzo uniwersalny typ wskaźnika. W konkretnej sytuacji wymagane jest jawne rzutowanie typu wskaźnika, co czasami może być przyczyną kłopotów.
Zastosujemy funkcję biblioteczną **qsort()** do posortowania tablicy liczb rzeczywistych (listing 15.2).Pokazana w programie funkcja porównująca **por()** ma dość nieczytelną postać. Inna implementacja funkcji porównującej może mieć bardziej czytelną postać:

```
int por( const void *w1, const void *w2)
{
  const double *p, *q;
  p = (const double*) w1;
  q = (const double*) w2;
         if (*p < *q)   return -1;
  else if (*p == *q) return  0;
  else                    return  1;
}
```

Listing 15.2. Funkcja biblioteczna qsort()

```
/* quick sort, liczby */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define NUM 5
void wprowadz_dane(double tab[], int n);
void drukuj_tab(const double tab[], int n);
int por(const void *w1, const void *w2);

int main(void)
{
 int i;
 double wart[NUM];
 wprowadz_dane(wart, NUM);
 printf("\nLiczby nieuporzadkowane\n");
 drukuj_tab(wart, NUM);
 qsort(wart, NUM, sizeof(double),por);
 printf("\n\nLiczby porzadkowane\n");
 drukuj_tab(wart, NUM);
 getche();
 return 0;
}

void wprowadz_dane(double tab[], int n)
 {
  printf("\nwprowadz liczby \n");
  for (int i = 0; i < n; i++)
   {
    printf("%d ", i);
    scanf("%lf", &tab[i]);
   }
 }

void drukuj_tab(const double tab[], int n)
{for (int i=0; i< n; i++)
       printf("\n%lf",tab[i]);
}

int por( const void *w1, const void *w2)
{ const double *p, *q;
  p = (const double*) w1;
  q = (const double*) w2;
  double x;
  x = (*p - (int) *p) - (*q - (int) *p);
  return ((x == 0.0) ? 0 : (x < 0.0) ? -1 : 1);
}
```

Oto przykładowy wynik działania programu z listingu 15.2 :

```
Wprowadz liczby
9
10.5
11.7
50
12

liczby nieuprzadkowane
  9.000000
10.500000
11.700000
50.000000
12.000000

liczby porzadkowane
 9.000000
10.000000
11.700000
12.000000
50.000000
```

Aby porównać wskazywane wartości, należy najpierw odczytać je z pamięci, co wymaga ***dereferencji*** wskaźników. Ponieważ wartości należą do typu **double**, konieczna jest ***dereferencja*** do typu **double**. Równocześnie jednak funkcja **qsort()** wymaga, aby funkcja porównująca przyjmowała wskaźniki typu **void.** Rozwiązaniem jest jawna konwersja typu wskaźnika:

```
const double *p, *q;
p = (const double*) w1;
q = (const double*) w2;
```

Widzimy, że funkcja **qsort()** i funkcja porównująca wykorzystują wskaźniki do **void** dla uzyskania elastyczności.

## 15.3. Funkcje biblioteki matematycznej - <math.h>

W standardowej bibliotece funkcji umieszczono szereg podstawowych funkcji matematycznych. Deklaracje i prototypy funkcji matematycznych znajdują się w pliku nagłówkowym **<math.h>.** Najważniejsze kategorie funkcji matematycznych to:

- Funkcje trygonometryczne
- Funkcje cyklometryczne
- Funkcje logarytmiczne
- Funkcje wykładnicze

Funkcje trygonometryczne wymagają argumentów (są to kąty) mierzonych w radianach ($360^o = 2\pi$ rad, 1 radian = $180/\pi = 57.296$ stopni).

Początkowo biblioteka funkcji matematycznych była zaprojektowana dla typów **double**, ale możemy też używać typów **float** i **long double**. Należy pamiętać, że biblioteka funkcji matematycznych jest cały czas rozwijana, pojawiają się nowe funkcje. Standard ANSI C99 wprowadził możliwość obsługi funkcji z argumentami zespolonymi (plik <complex.h>).
W tabeli zestawiono wybrane standardowe funkcje matematyczne ANSI C.

| Nr | Prototyp | Opis |
|---|---|---|
| 1 | double acos(double x) | zwraca arcus cosinus x (od 0 do π rad) |
| 2 | double asin(double x) | zwraca arcus sinus x (od –π/2 do π/2 rad) |
| 3 | double atan(double x) | zwraca arcus tangens x (od –π/2 do π/2 rad) |
| 4 | double atan2(double x, double y) | zwraca kąt którego tangens wynosi y/x (od –π do π rad) |
| 5 | double cos(double x) | zwraca cosinus x |
| 6 | double sin(double x) | zwraca sinus x |
| 7 | double tan(double x) | zwraca tangens x |
| 8 | double cosh(double x) | zwraca cosinus hiperboliczny z x |
| 9 | double sinh(double x) | zwraca sinus hiperboliczny z x |
| 10 | double tanh(double x) | zwraca tangens hiperboliczny z x |
| 11 | double exp(double x) | zwraca exp |
| 12 | double frexp(double v, int *pt_e) | rozbija wartość v na mantysę oraz potęgę dwójki |
| 13 | double ldexp(double x, int p) | zwraca 2 do potęgi p razy x |
| 14 | double log(double x) | zwraca logarytm naturalny z x |
| 15 | double log10(double x) | zwraca logarytm o podstawie 10 z x |
| 16 | double modf(double x, double *p) | rozbija x na część całkowitą i część ułamkową |
| 17 | double pow(double x, double y) | zwraca x do potęgi y |
| 18 | double sqrt(double x) | zwraca pierwiastek kwadratowy z x |
| 19 | double ceil(double x) | zwraca najmniejszą wartość całkowitą nie mniejszą niż x (zaokrągla w górę) |
| 20 | double fabs(double x) | zwraca wartość bezwzględną z x |
| 21 | double floor(double x) | zwraca największą wartość całkowitą nie większą niż x (zaokrągla w dół) |
| 22 | double fmod(double x, double y) | zwraca część ułamkową wartości x/y |

# ROZDZIAŁ 16

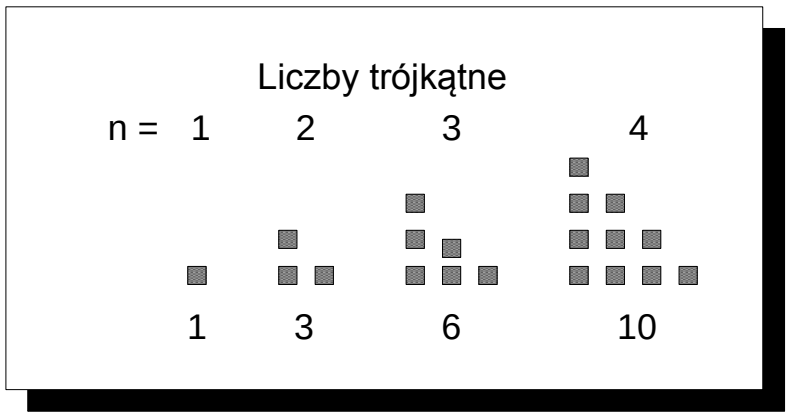
## REKURENCJA, GENERATORY LICZB PSEUDOLOSOWYCH

## 16.1. Wstęp

Gdy omawiane jest tak skomplikowane zagadnienie, jakim jest nauka programowania w języku C, natrafiamy na tematy, które nie są łatwe do zaklasyfikowania do pojedynczej, określonej kategorii. W tym rozdziale omówimy niektóre, wybrane tematy. Omówione zostaną trzy grupy zagadnień: rekurencje, generatory liczb pseudolosowych i sortowanie.

## 16.2. Rekurencje

W typowym programie, funkcje wywoływane są z funkcji **main()** lub z innych funkcji. Istnieje w języku C możliwość, że dana funkcja wywoła sama siebie. Funkcja, która wywołuje sama siebie jest nazywana ***funkcją rekurencyjną,*** sam proces nosi nazwę ***rekurencji*** (ang. *recursion*). Rekurencja ma wiele zastosowań praktycznych. Omawianie zagadnienia funkcji rekurencyjnych zaczniemy od analizy liczb trójkątnych. W starożytnej Grecji grupa filozofów pracująca pod przewodnictwem Pitagorasa niesłychanie ceniła liczby trójkątne. Liczby trójkątne tworzą ciąg nieskończony: 1,3,6,10,15,21,28,.....Jak łatwo zauważyć, n-ty wyraz ciągu otrzymamy dodając **n** do poprzedniego wyrazu. Jeżeli uznamy, że wartość liczbowa n-tego wyrazu jest liczbą prostych elementów, to z tych elementów daje się zbudować obiekt przypominający trójkąt prostokątny, tak jak to przedstawiono na rys. 16.1.




Rys.16.1. Model liczb trójkątnych

Wzór rekurencyjny pozwalający obliczyć n-ty wyraz ciągu liczb trójkątnych ma postać:

$$w_n = w_{n-1} + n$$

Można też obliczyć n-ty wyraz bezpośrednio:

$$w_n = \frac{n^2 + n}{2}$$

Analizując model, można zauważyć, że n-tą wartość otrzymujemy sumując kwadraty tworzące kolumny. Dla n = 4 mamy:

4 kwadraty w pierwszej kolumnie
3 kwadraty w drugiej kolumnie
2 kwadraty w trzeciej kolumnie
1 kwadrat w czwartej kolumnie

Zsumowanie 4 + 3 + 2 + 1 daje 10.
Napisanie programu sumującego kwadraty w kolumnach przy wykorzystaniu pętli jest już prostym zadaniem.

Listing 16.1. Liczby trójkątne, iteracja

```
#include <stdio.h>
#include <conio.h>
unsigned long oblicz(int );
int main(void)
{  int n;
   printf("\nPodaj numer wyrazu ciagu : ");
   scanf("%d",&n);
   printf("\nWyraz nr %d ma wartosc %ld",n,oblicz(n));
   getche();
   return 0;
}
unsigned long oblicz( int x)
{ unsigned long suma = 0;
  while (x > 0)
     { suma += x;
       --x;
     }
  return suma;
}
```

Obliczanie n-tego wyrazu ciągu liczb trójkątnych realizuje funkcja **oblicz():**

```
unsigned long oblicz( int x)
{ unsigned long suma = 0;
  while (x > 0)
     {suma += x;
      --x;
     }
  return suma;
}
```

Funkcja **oblicz()** sumuje elementy w kolumnie (rys.16.1), aby wyliczyć liczbę trójkątną. Pętla **while** dokonuje **n** przebiegów, dodając do zmiennej **suma** najpierw **n**, następnie **n-1**, aż do 1, kończąc sumowanie, gdy **n** osiągnie 0.
Obliczanie n-tego wyrazu ciągu metoda iteracyjną (w pętli) jest metodą bardzo oczywistą. Wiemy, że wyrazy ciągu są opisane wzorem rekurencyjnym, inaczej mówiąc n-ty wyraz to wyraz poprzedni plus **n**. Aby obliczyć n-ty wyraz możemy sumować tylko dwa składniki. Korzystając z naszego modelu (rys.16.1) ta suma to:

1) pierwsza kolumna, która ma wartość n
2) oraz suma wszystkich pozostałych kolumn

Program pokazany na kolejnym listingu wylicza n-ty wyraz ciągu liczb trójkątnych za pomocą funkcji rekurencyjnej.

Listing 16.2. Liczby trójkątne, rekurencja

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

unsigned long oblicz(int );

int main(void)
{  int n;
   printf("\nPodaj numer wyrazu ciagu : ");
   scanf("%d",&n);
   printf("\nWyraz nr %d ma wartosc %ld",n,oblicz(n));
   getche();
   return 0;
}

unsigned long oblicz (int x)
{ if (x == 1)
    return 1;
  else
    return (x + oblicz(x-1) );
}
```

Na początku podajemy numer wyrazu, który chcemy obliczyć. Funkcja **main()** wywołuje funkcje **oblicz()** z tym parametrem. Obliczanie n-tego wyrazu ciągu realizuje następujący fragment:

```
if (x == 1)
  return 1;
else
  return (x + oblicz(x-1) );
```

Jeżeli parametr jest równy 1, to funkcja zwraca 1 i program kończy działanie.

Gdy parametr jest większy od jeden, przyjmijmy, że n = 4, sytuacja jest całkiem odmienna. Następuje wywołanie funkcji **oblicz(4).**
W wyrażeniu instrukcji **return**:

```
return (x + oblicz(x-1) );
```

następuje ponowne wywołanie funkcji **oblicz(3).** Funkcja **oblicz()** wywołuje sama siebie, z tym tylko, że parametr funkcji ma inną wartość. Następuje powtórzenie sytuacji. W instrukcji **return** następuje ponowne wywołanie funkcji **oblicz(2**). Tym razem w instrukcji **return** mamy do czynienia z wywołaniem **oblicz(1),** a to jak wiemy kończy program. Być może ten opis działania funkcji rekurencyjnej jest niejasny. Możemy takie działanie opisać inaczej. Należy wyobrazić sobie, że mamy do czynienia z ciągiem wywołań funkcji – **oblicz1()** wywołuje **oblicz2(),** ta z kolei wywołuje **oblicz3(),** a ta wywołuje **oblicz4().** Gdy **oblicz4()** kończy działanie, program wraca do funkcji **oblicz3().** Gdy ona kończy działanie, program wraca do **oblicz2(),** gdy ta kończy działanie, program wraca do funkcji **oblicz1().** Rekurencja działa podobnie, z tym, że zamiast **oblicz1(), oblicz2(), oblicz3()** i **oblicz4()** mamy jedną i tą samą funkcję.
Jak działa w rzeczywistości program z funkcją rekurencyjną można zobaczyć, modyfikując funkcję rekurencyjną w powyższym listingu.

```
unsigned long oblicz (int x)
{ printf("\nWywolanie oblicz( %d)",x);
  if (x == 1)
     { printf("\n oblicz() = 1");
       return 1;}
  else
     { unsigned long wn =  x+ oblicz(x-1);
       printf("\n oblicz() = %ld", wn);
       return wn;}
}
```

Wewnątrz funkcji **oblicz()** umieszczono trzy funkcje **printf(),** które dają odpowiednie komunikaty w momencie, gdy funkcja **main()** wywołuje funkcję **oblicz()** i kiedy funkcja **oblicz()** wywołuje sama siebie. Dzięki tym komunikatom mamy możliwość śledzenia jak w rzeczywistości pracuje funkcja rekurencyjna. Po uruchomieniu programu ze zmodyfikowaną funkcja **oblicz()** otrzymamy następujące rezultaty:

```
Podaj numer wyrazu ciagu : 4
  Wywolanie oblicz(4)
  Wywolanie oblicz(3)
  Wywolanie oblicz(2)
  Wywolanie oblicz(1)
   oblicz() = 1
  oblicz() = 3
  oblicz() = 6
  oblicz() = 10
  Wyraz nr 4 ma wartość 10
```

Typowym przykładem stosowania funkcji rekurencyjnych jest program obliczający silnie liczby nieujemnej. Silnie dla n całkowitego, większego lub równego zeru, definiujemy następująco:

$$n! = 1 * 2 * 3 * .....n = \prod_{k=1}^{n} k$$

Zgodnie z konwencją 0! = 1. Możemy napisać:

$$n! = (n-1)!n$$

Kilka początkowych wartości funkcji silnia:

| n | n! |
|---|---|
| 0 | 1 |
| 1 | 1 |
| 2 | 2 |
| 3 | 6 |
| 4 | 24 |
| 5 | 120 |
| 6 | 720 |
| 7 | 5040 |
| 8 | 40320 |
| 9 | 362880 |
| 10 | 3628800 |

W przedstawionym programie, występuje funkcja **oblicz(),** wyliczająca silnie za pomocą pętli **while** (iteracyjnie).

Listing 16.3. Obliczanie silni, iteracja

```
#include <stdio.h>
#include <conio.h>
unsigned long oblicz(int );
int main(void)
{  int n;
   printf("\nPodaj liczbe : ");
   scanf("%d",&n);
   printf("\n Silnia liczby %d = %ld", n, oblicz(n));
   return 0;
}
unsigned long oblicz (int x)
{ unsigned long silnia = 1;
  while (x>0)   {
     silnia *= x;
     --x;
    }
  return silnia;
}
```

Po uruchomieniu mamy przykładowy wynik:

```
Podaj liczbe : 11
Silnia liczby 11 = 39916800
```

Ogólnie nasz program ogranicza argumenty funkcji silnia do przedziału 0 -15, ponieważ 15! przekracza 2 miliardy. Przy próbie liczenia 16! najczęściej przekroczymy zakres typu **long** na typowym PC-cie.
Wersja rekurencyjna obliczania silni korzysta z faktu, że n! = n* (n-1)!. W pokazanym programie należy odpowiednio zmienić funkcje **oblicz()**. Program rekurencyjny może mieć postać pokazaną na listingu.

Listing 16.4. Obliczanie silni, rekurencja

```
#include <stdio.h>
#include <conio.h>
unsigned long oblicz(int );
int main(void)
{  int n;
   printf("\nPodaj liczbe : ");
   scanf("%d",&n);
   printf("\n Silnia liczby %d = %ld", n, oblicz(n));
   getche();
   return 0;
}
unsigned long oblicz (int x)
{ if (x <= 1)
     return 1;
  else
     return ( x * oblicz(x-1) );
}
```

Pokazany program działa identycznie jak jego wersja iteracyjna.
Innym interesującym przykładem zastosowania funkcji rekurencyjnych jest zagadnienie obliczania wyrazów ciągu Fibonacciego. Jest to bardzo szczególny ciąg liczbowy (niektórzy nazywają go pięknym), którego początkowe liczby są następujące:

| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $F_n$ | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 |

Liczby Fibonacciego są liczbami całkowitymi, definiuje rekurencja:

$$F_0 = 0;$$
$$F_1 = 1;$$
$$F_n = F_{n-1} + F_{n-2} \quad \text{dla n} > 1$$

Należy opracować odpowiednią funkcje rekurencyjna. Nasza funkcja o nazwie **oblicz()** powinna obliczać liczby Fibonacciego w następujący sposób:

```
oblicz(0) = 0
oblicz(1) = 1
oblicz(n) = oblicz(n-1) + oblicz(n-2)
```

Poniżej pokazany jest listing programu obliczającego rekurencyjnie liczby Fibonacciego.

Listing 16.5. Obliczanie wyrazów ciągu Fibonacciego, rekurencja

```
#include <stdio.h>
#include <conio.h>
unsigned long oblicz(int );
int main()
{  int n;
   printf("\nPodaj liczbe : ");
   scanf("%d",&n);
   printf("\n Fibonacci(%d) = %ld", n, oblicz(n));
   getche();
   return 0;
}
unsigned long oblicz (int x)
{ if (x == 0  || x == 1)
     return x;
  else
     return ( oblicz(x-1) + oblicz(x-2) );
}
```

Po uruchomieniu programu otrzymuje następujący wynik:

```
Podaj liczbe : 30
Fibonacci(30) = 832040
```

Należy zauważyć, że liczby Fibonacciego bardzo szybko rosną, dlatego zastosowaliśmy typ danych **unsigned long**.
Funkcję rekurencyjną możemy wykorzystać bardzo efektywnie do napisania programu realizującego podnoszenie do potęgi naturalnej liczby rzeczywistej:

$$x^0 = 1$$
$$x^k = x \qquad \text{dla k} = 1$$
$$x^{-k} = 1/x^k \qquad \text{dla k dodatnich}$$
$$\text{x}^{\text{k}} = (x^{k-1})x \qquad \text{dla k dodatnich i nieparzystych}$$
$$x^k = (x^{k/2})(x^{k/2}) \qquad \text{dla k dodatnich i parzystych}$$

Postać programu do obliczania potęg pokazana jest na listingu.

Listing 16.6. Obliczanie wartości x do potęgi n, rekurencja

```
#include <stdio.h>
#include <conio.h>
double oblicz(double , int );
int main(void)
{  double x;
   int n;
   printf("\nPodaj liczbe : ");
   scanf("%lf",&x);
   printf("\nPodaj wykladnik calkowity : ");
   scanf("%d",&n);
   printf("\n %lf do potegi %d=%lf",x,n,oblicz(x,n));
   getche();
   return 0;
}
double oblicz (double x, int n)
{ double y;
  if (n < 0)        return (oblicz(1.0/x, -n));
  else if (n == 0)  return (1);
  else if (n == 1)  return (x);
  else if (n%2 == 0)
       {y = oblicz(x,n/2);
                    return (y*y);}
  else              return (x*oblicz(x, n-1) );
}
```

Po uruchomieniu tego programu mamy następujący wynik:

```
Podaj liczbe : 3
Podaj wykladnik calkowity : 3
3.000000 do potegi 3 = 27.000000
```

Inny przebieg da wynik:

```
Podaj liczbe : 5.2
Podaj wykladnik calkowity : 3
5.200000 do potegi 3 = 140.608000
```

Dowolne zadanie, jakie może być rozwiązane za pomocą rekurencji, może być rozwiązane iteracyjnie. Rozwiązanie rekurencyjne wybierane jest ze względu na elegancję i czytelność funkcji rekurencyjnej. Należy jednak pamiętać, że gdy chcemy zachować wydajność należy wybierać rozwiązanie iteracyjne. Rekurencja ma wiele wad. Wielokrotnie wywołuje funkcje. Konieczne jest przekazanie sterowania z punktu wywołania na początek wywołania funkcji. Do przechowywania pośrednich argumentów i wartości powrotnych używany jest wewnętrzny stos systemowy. Jeśli mamy do czynienia z dużą liczbą danych, może dojść do przepełnienia stosu.

## 16.3. Generowanie liczb losowych

Liczby losowe są bardzo przydatne w symulacjach komputerowych, testowaniu programów, w tworzeniu gier. Język C dostarcza odpowiednich narzędzi do tworzenia liczb, które zachowują się jak liczby losowe. Prawdziwe liczby losowe otrzymamy rzucając kostką sześcienną, w komputerze, korzystając z odpowiednich algorytmów możemy tworzyć jedynie tzw. ***liczby pseudolosowe***. Funkcje, dzięki którym otrzymujemy liczby pseudolosowe noszą nazwę generatorów liczb pseudolosowych.
Najbardziej zbadany algorytm służący do generowania liczb pseudolosowych oparty jest na następującej regule:

$$x_{j+1} = (ax_j + c) \bmod(m)$$

Generatory opisane powyższym wzorem nazywane są generatorami liniowymi kongruentnymi (LCG, ang. *linear congruential generators*). Symbol mod (m) lub modulo (m) oznacza, że wyrażenie stojące przed tym symbolem należy podzielić przez m i z otrzymanego wyniku zostawić tylko resztę, np. 5mod4 = 1.
Każda liczba pseudolosowa otrzymana z generatora LCG zależy od liczby poprzedniej, mnożnika **a**, przesunięcia **c** i modułu **m**. Jeżeli przesunięcie jest równe 0, to mamy inną regułę:

$$x_{j+1} = (ax_j) \bmod(m)$$

Generatory oparte na tej regule nazywamy generatorami multiplikatywnymi liniowymi kongruentnymi (MLCG, ang. *multiplicative linear congruential generators*).
Generatory MLCG są szybsze od generatorów LCG, ponieważ jednak generator MLCG nie wyprodukuje zera, częściej implementowane są generatory LCG. Generatory liczb pseudolosowych zawsze wyprodukują ciąg liczb okresowy. Długość takiego okresu zależy od implementacji. Należy jednak pamiętać, że do obliczeń i symulacji wybierać należy krótkie, w porównaniu z długością okresu, części całego ciągu.

Konstrukcja generatora liczb pseudolosowych jest prosta. Należy pamiętać jedynie o zakresach typów użytych do przechowywania liczb. Najczęściej generuje się liczby całkowite, stosujemy wtedy typ **unsigned long**. Przenośna wersja generatora liczb pseudolosowych może mieć postać pokazaną na listingu 16.7. Funkcja **psrand()** generuje liczby pseudolosowe. Kolejna liczba pseudolosowa generowana jest z poprzedniej. Wartość początkowa musi być podana przez użytkownika. W naszym przypadku jest to wartość równa jeden. Ta liczba początkowa nosi nazwę ***„ziarno”*** (ang. seed). Ponieważ ziarno musi być pamiętane, należy użyć zmiennej statycznej:

```
static unsigned long seed = 1;
```

Listing 16.7. Przenośny generator liczb pseudolosowych

```
/* generator LCG */
#include <stdio.h>
#include <conio.h>
#define ILE 10//ilosc wygenerownych liczb
unsigned long psrand();

int main()
{register int i;
 for ( i = 0; i<ILE; ++i)
     printf("\n%12ld",psrand());
 getche();
 return 0;
}

unsigned long psrand()
{static unsigned long seed = 1;
 seed = seed * 1103515245 + 12345;
 return  (seed/65536) % 32768;
}
```

Funkcja **psrand()** zwraca liczbę z zakresu od 0 do 32767. Uruchomieniu programu daje następujący wynik:

```
16838
 5758
10113
17515
31051
 5627
23010
 7419
16212
 4086
```

Na pierwszy rzut oka liczby te wyglądają na przypadkowe. Ponowne uruchomienie generatora powoduje otrzymanie takich samych liczb. Jest to oczywiste, ponieważ liczby pseudolosowe otrzymywane są na podstawie operacji arytmetycznych:

```
static unsigned long seed = 1;
seed = seed * 1103515245 + 12345;
return  (seed/65536) % 32768;
```

Łatwo zauważyć, że otrzymamy inne wyniki, gdy zmienimy wartość początkową ziarna, np. zmieniając jeden na pięć:

```
static unsigned long seed = 5;
```

Po uruchomieniu generatora, początkowe liczby wyglądają następująco:

```
18655
 8457
10616
31877
10193
```

Generatory liczb pseudolosowych maja wbudowane mechanizmy pozwalające nadać ziarnu dowolną wartość.

W poważnych symulacjach wymagane są generatory liczb pseudolosowych o doskonałych parametrach, szybkie i o długim okresie. Istnieje bogata literatura na ten temat. Do celów testowych każda implementacja języka C zawiera funkcje biblioteczne służące do generacji liczb pseudolosowych. W pliku nagłówkowym **<stdlib.h>** znajdują się funkcje, makra i stałe związane z produkowaniem liczb pseudolosowych:

```
rand()
random()
randomize()
srand()
RAND_MAX
```

## 16.4. Generator liczb pseudolosowych rand()

Bardzo wygodnym i bardzo często używanym do najrozmaitszych zadań generatorem liczb pseudolosowych jest biblioteczny generator **rand().** Deklaracja generatora jest następująca:

```
int rand(void)
```

W wyniku wywołania generatora zwracana jest liczba pseudolosowa z zakresu od 0 do RAND_MAX. Stała RAND_MAX jest zdefiniowana w pliku **<stdlib.h>** i zwykle jest równa INT_MAX (największej możliwej liczbie typu **int**).

Listing 16.8. Generator liczb pseudolosowych – rand()

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

int main()
{
   printf("dziesiec liczb pseudolosowych\n\n");
   for(int i = 0; i < 10; i++)
       printf("%10d\n", rand() );
   getche();
   return 0;
}
```

Po uruchomieniu tego programu mamy następujący wynik:

```
dziesiec liczb pseudolosowych
    346
    130
  10982
   1090
  11656
   7117
  17595
   6415
  22948
  31126
```

Widzimy, że liczbę losową otrzymamy przez proste przypisanie:

```
ps = rand();
```

a wartości wytwarzane bezpośrednio przez **rand()** są zawsze w zasięgu:

$$0 \leq ps \leq RAND\_MAX$$

Bardzo często potrzebujemy innego zakresu generowanych liczb, np. przy symulacji rzutów kostką sześcienną otrzymujemy tylko liczby 1, 2, 3, 4, 5 i 6. Aby wygenerować zadany zakres liczb, należy dokonać skalowania:

```
ps = a + rand() % b
```

**a** jest wartością przesunięcia, **b** jest czynnikiem skalowania (jest równy szerokości zakresu kolejnych liczb całkowitych). Aby symulować rzut kostką sześcienną nasze skalowanie ma postać:

```
ps = 1 + rand() % 6
```

Gdy chcemy mieć zakres liczb pseudolosowych od 0 do 99 to używamy instrukcji:

```
ps = rand() % 100
```

## 16.5. Inicjalizowanie generatora liczb pseudolosowych srand()

Jak już wiemy, proste uruchamianie generatora liczb pseudolosowych daje w wyniku taki sam ciąg liczb. Dzieje się tak, dlatego, że generator startuje zawsze z tej samej wartości ziarna. Jeżeli będziemy zmieniali ziarno, będziemy mogli otrzymywać różne ciągi liczbowe. W języku C jest specjalny mechanizm do zmiany ziarna generatora **rand().** Służy do tego funkcji **srand().** Deklaracja funkcji **srand()** jest następująca:

```
void srand(unsigned seed)
```

Funkcja **srand()** potrzebuje odpowiedniego parametru. W tym celu wykorzystuje się zegar systemowy. Biblioteka ANSI C zawiera funkcje **time(),** która zwraca czas systemowy. Zwracana wartość jest liczbą i jej wartość

zmienia się w czasie. W programie pokazanym na listingu 16.9 symulującym rzuty kostką sześcienną przez dwóch graczy wykorzystaliśmy funkcje **srand()** i zegar systemowy.

Listing 16.9. Liczby pseudolosowych – rand(), srand()

```
/* generator rand(),srand(), kostka */
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include <conio.h>
int main()
{int gr1, gr2;
 srand(time(NULL));
 for(int i = 0; i < 4; i++)
  { gr1 = 1 + rand() % 6;
    gr2 = 1 + rand() % 6;
    printf("\nrzut %2d gr1=%2d  gr2=%2d",i+1,gr1,gr2);
  }
 return 0;
}
```

Po uruchomieniu programu mamy następujący wynik:

```
rzut 1  gr1 =   5   gr2 =   5
rzut 2  gr1 =   3   gr2 =   5
rzut 3  gr1 =   5   gr2 =   2
rzut 4  gr1 =   4   gr2 =   2
```

Pętla **for** powtarzana jest cztery razy, za każdym razem losowane są dwie liczby pseudolosowe z zakresu od 1 do 6. Każdorazowo po uruchomieniu programu otrzymamy inny wynik – mamy rzeczywistą symulacje gry w kości. Za każdym razem zmienia się ziarno w generatorze **rand().** Tą sytuacje powoduje użycie wyrażenia takiego jak:

```
srand(time(NULL));
```

Funkcja **srand()** jest wywoływana tylko raz w programie, aby dać pożądany rezultat losowy. Nie ma potrzeby wywoływania jej wielokrotnie. Do programu należy włączyć plik nagłówkowy **<time.h>** w którym umieszczona jest funkcja **time().**

## 16.6. Generator liczb pseudolosowych random()

W bibliotece standardowej **<stdlib.h>** znajduje się jeszcze jeden generator liczb pseudolosowych o nazwie **random().** Według dokumentacji technicznej, może on występować w postaci makrodefinicji i funkcji, a deklaracje mają postać:

- makro: random(num)
- funkcja: int random(int num)

Po wywołaniu **random()** zwracana jest liczba pseudolosowa z zakresu pomiędzy 0 i (num – 1). W zasadzie generator jest przenaszalny na systemach DOS i WINDOWS.

## 16.7. Makrodefinicja randomize()

Makrodefinicja **randomize()** służy do inicjalizacji generatorów liczb pseudolosowych. Jej deklaracja ma postać:

```
void randomize(void)
```

Ponieważ to makro wywołuje funkcję **time(),** należy dołączyć plik nagłówkowy **<time.h>.** Makro stosowane jest w zasadzie w systemach DOS i WINDOWS. Praktyczne zastosowanie generatora **random()** i makra **randomize()** pokazano w programie, który produkuje dziesięć licz pseudolosowych z zakresu od 0 do 9.

Listing 16.10. Liczby pseudolosowych, random(), randomize()

```
/* generator random(), randomize() */
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include <conio.h>
int main()
{  randomize();
   printf("\n10 liczb losowych z zakresu 0-9\n");
   for (int i = 0; i < 10; i++)
   printf("\n %3d  ps =  %d",i+1, random (10));
   getche();
   return 0;
}
```

Po uruchomieniu programu otrzymujemy następujący wynik:

```
10 liczb losowych z zakresu 0-9
1  ps  = 7
2  ps  = 5
3  ps  = 3
4  ps  = 3
5  ps  = 9
6  ps  = 8
7  ps  = 2
8  ps  = 0
9  ps  = 9
10 ps  = 6
```

Oczywiście, dzięki zastosowaniu makra **randomize(),** każde ponowne uruchomienie programu da inny wynik.

## 16.8. Ustalanie zakresów generowanych liczb pseudolosowych

W wielu przypadkach potrzebujemy liczb pseudolosowych z konkretnego zakresu. Jak już wiemy, najbardziej uniwersalnym sposobem generowania takich liczb jest użycie funkcji **rand().** Funkcja ta dostarcza liczbę całkowita z przedziału [0, RAND_MAX]. Stała RAND_MAX powinna być nie mniejsza niż maksymalna wartość liczby typu **int**, czyli 32767. Aby otrzymać liczbę pseudolosową z przedziału [0, 1] należy wygenerowaną przez **rand()** liczbę podzielić przez RAND_MAX + 1. Taką liczbę należy pomnożyć przez n + 1, aby otrzymać liczbę z przedziału [0, n]. Mamy trzy przypadki:

$$ps = rand() \qquad [0, \mathrm{RAND\_MAX}]$$
$$ps = rand()/(RAND\_MAX + 1.) \qquad [0,1]$$
$$ps = rand()/(RAND\_MAX + 1.)*(n+1) \qquad [0, n]$$

Praktyczna realizacja generatora liczb pseudolosowych z zakresu [0,n] pokazana jest na listingu 16.11. Generowanie liczb pseudolosowych wykonywane jest za pomocą funkcji **rg_n().** Funkcja **rg_n()** ma jeden parametr – zakres generowanych liczb.

Listing 16.11. Liczby pseudolosowych, zakres [0,n]

```
/* funkcja pseudolosowa */
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>
#define ZA 10      //zakres [0, ZA]
int rg_n(int);
int main()
{for (int i = 0; i<20; i++)
     printf("\nZakres [0, n]  ps =  %d", rg_n(ZA));
 return 0;
}
int rg_n(int n)   //zakres [0,n]
{int ps;
 static int flag = 1;
 time_t date;
 if(flag)
   { srand(time(&date));
     flag = 0;
   }
 ps = rand()/(RAND_MAX + 1.0)*(n+1);
 return (ps);
}
```

Ta funkcja wykorzystuje funkcje **srand(),** aby losowanie było zmienne w czasie

(ziarno obliczane jest na podstawie wskazań zegara systemowego). Program wyświetla 20 liczb pseudolosowych z zakresu podanego przez użytkownika.
Praktyczna realizacja generatora liczb pseudolosowych z zakresu [0,1] pokazana jest na listingu 16.12. Generowanie liczb pseudolosowych wykonywane jest za pomocą funkcji **rg_1().** Funkcja **rg_1()** jest typu **float** i nie ma parametrów wejściowych – zakres generowanych liczb zawsze jest w przedziale [0, 1]. Ta funkcja wykorzystuje funkcje **srand(),** aby losowanie było zmienne w czasie (ziarno obliczane jest na podstawie wskazań zegara systemowego).

Listing 16.12 Liczby pseudolosowych, zakres [0, 1]

```
/* funkcja pseudolosowa [0,1] */
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>
float rg_1();
int main()
{ for (int i = 0; i<20; i++)
     printf("\nZakres [0, 1]  ps =  %f", rg_1());
  getche();
  return 0;
}
float rg_1()   //zakres [0,1]
{float ps;
 static int flag = 1;
 time_t date;
 if(flag)
   { srand(time(&date));
     flag = 0;
   }
 ps = (float) rand()/(RAND_MAX + 1.0);
 return (ps);
}
```

## 16.9. Szacowanie wartości liczby Pi

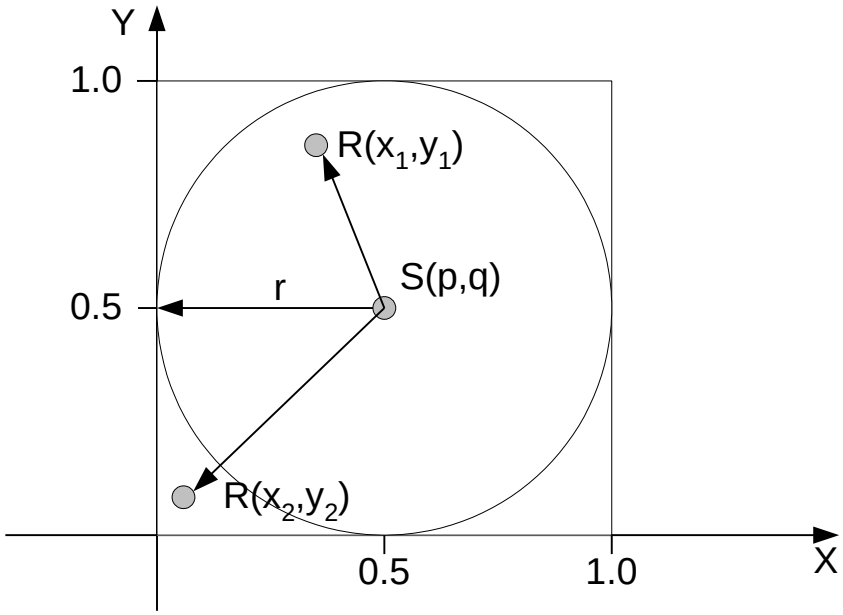
Zastosujemy omawiane generatory liczb pseudolosowych do obliczenia przybliżonej wartości liczby **Pi**. Do oszacowania wykorzystamy metody rachunku prawdopodobieństwa i metody symulacji komputerowych. Niech w kwadrat jednostkowy wpisane będzie koło (rys.16.2).

Za pomocą generatora liczb pseudolosowych losujemy współrzędne punktów. Wylosowany punkt może leżeć wewnątrz koła lub nie. Należy obliczyć stosunek liczby punktów wylosowanych w kole (L) do liczby wszystkich losowań (N).

Ten stosunek równy jest stosunkowi powierzchni kwadratu do powierzchni

koła:

$$\frac{L}{N} = \frac{\pi d^2}{4d^2} = \frac{\pi}{4}$$




Rys.16.2. Metoda szacowania liczby Pi

Program obliczający przybliżoną wartość stałej **Pi** pokazany jest na listingu 16.13. Funkcja **gr_1()** generuje liczby pseudolosowe z przedziału [0,1]. Do tego celu wykorzystano generator **rand():**

```
ps = (float) rand()/(RAND_MAX + 1.0);
```

Aby zapewnić losowanie zmienne w czasie wykorzystano funkcję **srand():**

```
srand(time(&date));
```

Funkcja **srand()** powinna być wywołana tylko jeden raz, dlatego mamy warunek:

```
static int flag = 1;
if(flag)
   { srand(time(&date));
     flag = 0;
   }
```

Listing 16.13. Oszacowanie liczby Pi

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>
float rg_1();
int main()
{ float pi, x, y, r2;
  int n;
  int l = 0;
  printf("\nPodaj liczbe losowan: ");
  scanf("%d",&n);
  for (int i = 1; i<=n; i++)
    {x = rg_1();
     y = rg_1();
     r2 = (x-0.5)*(x-0.5) + (y-0.5)*(y-0.5);
     if (r2 <= 0.25) l++;
    }
  pi = 4.0*l/n;
  printf("\ndla n= %d oszacowanie pi = %f", n,pi);
  return 0;
}
float rg_1()   //zakres [0,1]
{float ps;
 static int flag = 1;
 time_t date;
 if(flag)
   { srand(time(&date));
     flag = 0;
   }
 ps = (float) rand()/(RAND_MAX + 1.0);
 return (ps);
}
```

Dla każdego punktu należy wylosować jego współrzędne (x,y):

```
x = rg_1();
y = rg_1();
```

Kolo wpisane w kwadrat jednostkowy (rys.16.2) ma środek w punkcie S o współrzędnych (0.5, 0.5). Dla wylosowanego punktu należy obliczyć jego odległość od środka koła (dla przyspieszenia obliczeń obliczamy kwadrat odległości):

```
r2 = (x-0.5)*(x-0.5) + (y-0.5)*(y-0.5);
```

a następnie sprawdzić, czy punkt leży wewnątrz okręgu:

```
if (r2 <= 0.25) l++;
```

Jeżeli kwadrat odległości wylosowanego punktu od środka koła jest mniejszy lub równy kwadratowi promienia **r** (w kole jednostkowym jest to 0.25) to

zwiększamy licznik punktów leżących wewnątrz koła. Ostateczne oszacowanie liczby Pi:

```
pi = 4.0*l/n;
```

jest wyświetlane na ekranie monitora. Ponieważ mamy do czynienia z symulacjami komputerowymi, za każdym razem dostaniemy inny wynik. Należy także spodziewać się, że dokładność oszacowania będzie większa dla większej liczby losowań. Przykładowe rezultaty wykonania programu pokazano w tabeli.

| Liczba losowań N | Oszacowanie Pi |
|---|---|
| 100 | 3.200000 |
| 500 | 3.128000 |
| 1000 | 3.216000 |
| 5000 | 3.140800 |
| 10000 | 3.152800 |
| 20000 | 3.147000 |

## 16.10. Sortowanie

Sortowanie jest prawdopodobnie najczęściej wykonywanym zadaniem przez komputery. Kiedy tylko jest stworzony większy zbiór danych, wcześniej czy później pojawi się zagadnienie jego sortowania. Ponieważ sortowanie jest tak ważne, jest ono przedmiotem wielu badań i specjalistycznych publikacji. Jest wiele metod sortowania. Jedna z najprostszych metod jest ***sortowanie bąbelkowe***. Program pokazany jest na listingu 16.14.
Przykładowe uruchomienie programu da następujący wynik:

```
Podaj liczbe danych: 7
Dane nieposortowane
20 74 65 87 35 66 68
Dane posortowane
87 74 68 66 65 35 24
```

W programie użyto trzech funkcji użytkowych:

```
void dane_tab(int tab[], int n);
void sort_dane(int tab[], int n);
void drukuj_tab(const int tab[], int n );
```

Funkcja **dane_tab()** wypełnia tablice liczbami pseudolosowymi, otrzymanymi z generatora **random().** Funkcja **sort_tab()** sortuje tablicę liczb w porządku malejącym, korzystając z metody sortowania bąbelkowego. Funkcja **drukuj_tab()** drukuje elementy tablicy w ośmiu kolumnach.

Listing 16.14. Sortowanie bąbelkowe

```
//wersja z funkcja sortowania
```

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>
#define ROZ 50    //maksymalna ilosc danych
#define ZA 100    //zakres generatora [0,ZA-1]
void dane_tab(int tab[], int n);
void sort_dane(int tab[], int n);
void drukuj_tab(const int tab[], int n );
int main()
{  int n,tab[ROZ];
   printf("\nPodaj liczbe danych: ");
   scanf("%d",&n);
   dane_tab(tab, n);
   puts("Dane niesortowane");
   drukuj_tab(tab,n);
   sort_dane(tab,n);
   puts("Dane sortowane");
   drukuj_tab(tab,n);
   getche();
   return 0;
}
void dane_tab(int tab[],int n) //produkuje dane losowe
{ randomize();
  for (int i = 0; i<n; i++)
   tab[i] = random(ZA);
}
void sort_dane(int tab[], int n)  //bubble sort
{ int temp;
   for (int i = 0; i < n-1;  ++i)
     for (int j = n-1; i < j; --j)
       if (tab[j-1] < tab[j])
         {temp = tab[j-1];
          tab[j-1] = tab[j];
          tab[j] = temp;
         }
}
void drukuj_tab(const int tab[],int n )
{for (int i=0; i<n; i++)
  {  printf(" %5d",tab[i]);
     if (i % 8 == 7)    putchar('\n');
  }
  if (i % 8 != 0)  putchar('\n');
}
```

Sortowanie bąbelkowe wymaga kilku przejść przez tablicę. Chcemy mieć liczby uporządkowane malejąco (oczywiście możemy także wybrać porządek rosnący). W każdym przejściu kolejne pary elementów są porównywane. Jeżeli para

znajduje się w porządku malejącym lub ich wartości są jednakowe, elementy są pozostawiane na swoich miejscach. Jeżeli para znajduje się w porządku rosnącym, jej wartości są zamieniane miejscami. Zazwyczaj sortowanie wykonywane jest w zagnieżdżonych pętlach **for.** Test dwóch sąsiednich elementów wykonywany jest w pętli wewnętrznej. Jeżeli sąsiednie elementy nie są uporządkowane, następuje zamiana ich miejsc. Zamiana wykonywana jest w następującym fragmencie kodu:

```
temp    = tab[i];
tab[i]  = tab[j];
tab[j]  = temp;
```

W pierwszej instrukcji zmienna **temp** czasowo przechowuje wartość **tab[i].** W następnej instrukcji ta wartość jest zamieniona przez wartość **tab[j**]. W ostatniej instrukcji wartość **tab[j]** jest zamieniona przez wartość **tab[i],** która teraz jest przechowywana w zmiennej **temp**. Przestawianie dwóch elementów tablicy w algorytmie sortowania bąbelkowego jest istotną autonomiczna operacją. Zgodnie z zasadą „dziel i rządź" spróbujemy wydzielić ten element i umieścić go w osobnej funkcji. W porównaniu z poprzednią wersją algorytmu sortowania bąbelkowego, jedyna zmiana algorytmu dotyczy funkcji sortującej, zmieniono także porządek na rosnący. Program pokazany jest na listingu 16.15.
Funkcja **sort_dane()** sortuje wskazaną tablicę, gdy spełniony jest warunek, wywoływana jest funkcja **przestaw()**, aby przestawić miejscami dwa elementy tablicy **ta[j]** i **ta[j+1]:**

```
if (ta[j] > ta[j+1])
   przestaw (&ta[j], &ta[j+1] );
```

Wiemy, że funkcja przestaw nie ma prawa dostępu do poszczególnych elementów tablicy w funkcji **sort_dane().** Wiemy również, że te elementy mogą być przekazane do funkcji **przestaw()** przez referencję – jawnie zostanie przekazany każdy z adresów elementów tablicy. Mimo, że cała tablica standardowo dostarczana jest przez referencję, poszczególne jej elementy przekazywane są przez wartość. Z tego powodu funkcja **sort_dane()** wykorzystuje operator pobrania adresu (**&**) dla każdego elementu tablicy przesyłanej do funkcji **przestaw()** (wywołanie funkcji **przestaw()):**

```
przestaw (&ta[j], &ta[j+1] );
```

Listing 16.15. Sortowanie bąbelkowe, inna implementacja

```
//wersja z funkcja sortowania i przestaw
#include <stdlib.h>
```

```
#include <stdio.h>
#include <conio.h>
#include <time.h>
#define ROZ 50    //maksymalna ilosc danych
#define ZA 100    //zakres generatora [0,ZA-1]

void dane_tab(int tab[], int n);
void sort_dane(int *, int );
void drukuj_tab(const int tab[], int n );
int main()
{  int n,tab[ROZ];
   printf("\nPodaj liczbe danych: ");
   scanf("%d",&n);
   dane_tab(tab, n);
   puts("Dane niesortowane");
   drukuj_tab(tab,n);
   sort_dane(tab,n);
   puts("Dane sortowane");
   drukuj_tab(tab,n);
   return 0;
}
void dane_tab(int tab[],int n) //produkuje dane losowe
{ randomize();
  for (int i = 0; i<n; i++)
    tab[i] = random(ZA);
}
void sort_dane(int *ta, int n)  //bubble sort
{void przestaw(int *, int *);
 for (int i = 0; i < n-1; i++)
   for (int j = 0; j <n-1;j++)
     if (ta[j] > ta[j+1])
       przestaw (&ta[j], &ta[j+1] );
}
void przestaw(int *ws1, int * ws2)
{ int temp = *ws1;
  *ws1 = *ws2;
  *ws2 = temp;
}
void drukuj_tab(const int tab[], int n )
{for (int i=0; i<n; i++)
  {  printf(" %5d",tab[i]);
     if (i % 8 == 7)    putchar('\n');
  }
 if (i % 8 != 0) putchar('\n');
}
```

Funkcja **przestaw()** otrzymuje **ta[j]** za pomocą zmiennej wskaźnikowej **ws1**:

```
void przestaw(int *ws1, int * ws2)
{
```

```
 int temp = *ws1;
 *ws1 = *ws2;
 *ws2 = temp;
}
```

Z powodu ukrywania informacji, funkcja **przestaw()** nie może posłużyć się nazwą **ta[j],** ale może użyć ***ws1**, który jest jej synonimem.
Bardzo podobną metodą sortowania jest metoda nazywana ***metodą sortowania przez proste wybieranie***. Algorytm tej metody jest bardzo prosty:

1. Tablica do sortowania jest przeszukiwana od pierwszego elementu do ostatniego, znajdowany jest element największy a jego indeks jest zapamiętany.
2. Zamienia się ten element z pierwszym elementem. Element największy znajduje się na początku tablicy, element, który znajdował się na początku tablicy wędruje na miejsce znalezionego największego elementu.
3. Tablica do sortowania jest przeszukiwana od drugiego elementu, znajdowany jest kolejny największy element a jego indeks jest zapamiętany.
4. Kolejno powtarzamy punkt drugi i trzeci aż zostanie ostatni element tablicy.

Listing 16.16 przedstawia realizację algorytmu sortowania przez proste wybieranie. W zasadzie kod programu jest prawie identyczny z kodem programu do sortowania bąbelkowego. Jedyna różnica występuje w kodzie funkcji sortującej **sort_dane().** Zakładamy, że sortujemy dane, które są elementami tablicy jednowymiarowej ( każdy element ma swój indeks).

Listing 16.16. Sortowanie przez proste wybieranie

```
//Sortowanie przez proste wybieranie
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>
#define ROZ 50    //maksymalna ilosc danych
#define ZA 100    //zakres generatora [0,ZA-1]

void dane_tab(int tab[], int n);
void sort_dane(int *, int );
void drukuj_tab(const int tab[], int n );
int main()
{  int n,tab[ROZ];
   printf("\nPodaj liczbe danych: ");
   scanf("%d",&n);
   dane_tab(tab, n);
   puts("Dane niesortowane");
   drukuj_tab(tab,n);
   puts("Dane w trakcie sortowania");
   sort_dane(tab,n);
```

```
    puts("Dane sortowane");
    drukuj_tab(tab,n);
    getche();
    return 0;
}
void dane_tab(int tab[],int n) //produkuje dane losowe
{randomize();
  for (int i = 0; i<n; i++)
   tab[i] = random(ZA);
}
void sort_dane(int *ta, int n)  //proste wybieranie
{int kmax;
 void przestaw(int *, int *);
   for (int i = 0; i < n-1; i++)
    {kmax = i;
     for (int j = i+1; j <n;j++)
        if (ta[j] > ta[kmax]) kmax = j;
     przestaw (&ta[kmax], &ta[i] );
     drukuj_tab(ta,n);
    }
}
void przestaw(int *ws1, int * ws2)
{int temp = *ws1;
 *ws1 = *ws2;
 *ws2 = temp;
}
void drukuj_tab(const int tab[], int n )
{for (int i=0; i<n; i++)
  {printf(" %5d",tab[i]);
   if (i % 8 == 7) putchar('\n');
  }
 if (i % 8 != 0) putchar('\n');
}
```

Aby pokazać proces sortowania, w funkcji **sort_dane()** umieszczono funkcję **drukuj_tab(),** na ekranie pokazywane są elementy tablicy po każdym przejściu pętli.
Metodę sortowania bąbelkowego i metodę sortowania przez proste wybieranie, jak widać jest łatwo zakodować, ale technika ta jest nieefektywna. Dla sortowania małych zbiorów nie ma to większego znaczenia, ale do sortowania dużej ilości danych należy używać znacznie szybszych metod sortowania (polecana jest metoda sortowania szybkiego (*quick sort),* omawiana już funkcja biblioteczna **qsort()** ).

Przykładowy wynik działania programu może mieć postać:

```
Podaj liczbe danych : 6
Dane niesortowane
   99   16   0   82   9   57
Dane w trakcie sortowania
   99   16   0   82    9   57
   99   82   0   16    9   57
   99   82   57   16    9    0
   99   82   57   16    9    0
   99   82   57   16    9    0
Dane sortowane
   99   82   57   16    9    0
```

## 16.11. Losowe wybieranie elementów

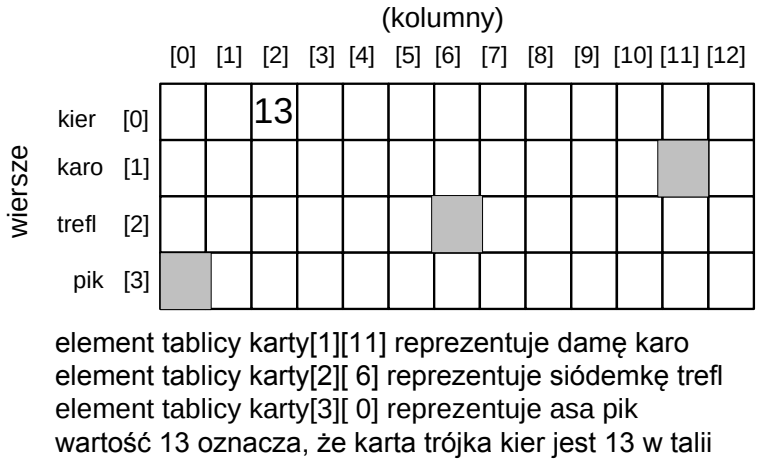
Algorytmy tasowania i wybierania kart wymagają dobrych generatorów liczb pseudolosowych i wydajnych metod odszukiwania konkretnego elementu. Podczas projektowania algorytmu tasowania możemy napotkać wiele subtelnych problemów związanych np. z szybkością ustalania rozkładu kart. Opiszemy prosty program, który symuluje tasowanie talii 52 kart do gry a następnie wyświetla rozdanie 13 kart każdemu z czterech graczy. Zakładamy, że w talii mamy 52 karty. Każda karta ma określony kolor (mamy cztery kolory : kier, karo, trefl i pik) oraz wartość (dla każdego koloru mamy 13 wartości: as, dwójka, trójka, czwórka, piątka, szóstka, siódemka, ósemka, dziewiątki, dziesiątka, walet, dama i król). Do przedstawienia kart w talii wykorzystamy tablicę dwuwymiarową: poszczególne wiersze odpowiadają kolorom, kolumny odpowiadają wartościom kart. Model zmiennej tablicowej **int karty[ ][ ]** pokazany został na rysunku 16.3. Jest to tablica o rozmiarze 4x13. Indeksowanie wierszy ma postać:

indeks 0 – kier
indeks 1 – karo
indeks 2 – trefl
indeks 3 – pik

Podobnie indeksujemy wartości poszczególnych kart (kolumny):

indeks 0 – as
indeks 1 – dwójka
indeks 2 – trójka
indeks 3 – czwórka
indeks 4 – piątka
indeks 5 – szóstka
indeks 6 – siódemka
indeks 7 – ósemka
indeks 8 – dziewiątka
indeks 9 – dziesiątka

indeks 10 – walet
indeks 11– dama
indeks 12 - król




Rys. 16.3. Dwuwymiarowa tablica reprezentująca karty w talii

Tasowanie kart polega na przestawianiu kolejności kart w talii. Po zakończeniu procesu tasowania kart, mamy ustaloną kolejność kart w talii. Oznacza to, że każda karta w talii ma unikalny numer związany z jej pozycją w talii. Symulację tasowania zrealizujemy następująco. Na początku wszystkie elementy tablicy **karty[wiersz][kolumna]** są inicjalizowane zerami. Następnie losowana jest karta, która będzie pierwszą w talii. Za pomocą generatora liczb pseudolosowych losujemy liczbę z zakresu 0 - 3 (inaczej mówiąc ustalamy losowo kolor karty, czyli ustalamy wartość wiersza). Następnie generujemy liczbę pseudolosową z zakresu 0 – 12, czyli losujemy wartość karty, co oznacza, że ustalamy wartość kolumny. Tak wybranemu elementowi tablicy nadajemy wartość 1 (karta jest pierwsza w talii). Następnie losujemy kolor i wartość karty, która będzie druga w talii. Proces powtarzamy tak długo aż ustalimy kolejność 52 kart. Może się zdarzyć, że wylosowane wartości indeksów reprezentują kartę, która już została wybrana. Możemy to sprawdzić, ponieważ w takim przypadku wybrany element będzie miał wartość różną od zera. W takim przypadku unieważniamy to losowanie i losujemy ponownie indeksy. Zawsze istnieje, (co prawda bardzo niewielkie) niebezpieczeństwo, że algorytm może być wykonywany w nieskończoność, jeżeli będą wybierane wciąż te same karty. Program symulujący tasowanie kart pokazany jest na listingu 16.17.

Listing 16.17. Tasowanie talii kart

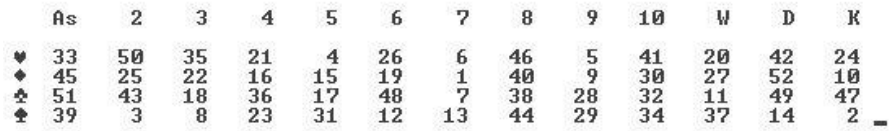
```
#include <stdio.h>
#include <conio.h>
```

```
#include <stdlib.h>
#include <time.h>
const char kolor[4] = {'\003', '\004', '\005','\006'};
void tasuj  ( int [ ][13]);
void talia  ( int [ ][13]);

int main()
{int karty [4][13] = { 0 };
 srand (time(0));
 tasuj ( karty );
 talia( karty );
 getche();
 return 0;
} //------- koniec int main() -----------

void tasuj ( int tkarty[][13])
{int kolor, walor;
 for (int karta = 1; karta <= 52; karta++)
  {
    do {kolor = rand() % 4;
        walor = rand() % 13;
       } while (tkarty[kolor][walor] != 0 );
    tkarty[kolor][walor] = karta;
  }
}   // ------ koniec tasuj() --------

void talia( int tkarty [][13])
{ printf("\n    As    2    3    4    5    6");
  printf("       7    8    9   10    W    D    K\n");
   for (int w = 0; w <=3; w++)
        { printf("\n %c", kolor[w]);
          for (int k = 0; k <= 12; k++)
             printf (" %3d ", tkarty[w][k]);
        }
}    // ---- koniec talia() --------
```

W programie mamy dwie funkcje pomocnicze:

```
void tasuj  ( int [ ][13]);
void talia  ( int [ ][13]);
```

Parametrem tych funkcji jest tablica dwuwymiarowa. Do wybierania kolejności kart służy funkcja **tasuj(),** funkcja **talia()** służy do wyświetlenia dwuwymiarowej tablicy reprezentującej karty w tali, wartości elementów tablicy są numerami kart w talii.
Po uruchomieniu tego programu mamy następujący wynik:




Rys. 16.4 Wynik działania programu tasującego talie.

Jak widać z wydruku, pierwszą kartą w talii (w tym przebiegu losowania) jest siódemka karo, drugą jest król pik, itd. Rozkład kart po każdym losowaniu jest inny, ponieważ wywoływana jest funkcja:

```
srand (time(0));
```

która zmienia parametry generatora liczb pseudolosowych, korzystając zegara systemowego. Ustalanie kolejności kart w talii realizuje następujący fragment funkcji **tasuj():**

```
for (int karta = 1; karta <= 52; karta++)
  {
    do {
        kolor = rand() % 4;
        walor = rand() % 13;
       } while (tkarty[kolor][walor] != 0 );
    tkarty[kolor][walor] = karta;
  }
```

W pętli **for** wybieramy kolejne pozycje (od 1 do 52). Pętla **do...while** wybiera losowo kolor karty i jej wartość. Sprawdzamy także, czy karta nie została już wybrana wcześniej (czy wartość odpowiedniego elementu tablicy jest równa 0).
Funkcja **talia()** realizuje wyświetlanie tablicy reprezentującej karty w potasowanej talii.
Do wyświetlenia symbolu koloru karty wykorzystano znaki ze zbioru ASCII. Jak wiadomo, każdy znak tego zbioru jest identyfikowany za pomocą unikatowej liczby. Kolory kart zapisaliśmy w tablicy kolor[]:

```
const char kolor[4] = {'\003', '\004', '\005','\006'};
```

Na przykład liczba 3 przedstawia symbol kiera. Aby wyświetlić ten znak możemy np. skorzystać z funkcji **putchar(),** pamiętając, że po znaku \ umieszcza się przyporządkowaną mu liczbę w postaci trzycyfrowej:

```
putchar ('\003');
```

Zestaw znaków symbolizujących kolory kart jest następujący:

| | |
|---|---|
| kier | '\003' |
| karo | '\004' |
| trefl | '\005' |
| pik | '\006' |

W naszej funkcji znaki kolorów kart drukuje instrukcja:

```
        printf("\n %c", kolor[w]);
```

W momencie, gdy już mamy potasowane karty, należy je rozdać czterem graczom. Program realizujący to zadanie pokazany jest na listingu 16.18.
Symulację rozdania kart czterem graczom (oznaczonym, jako N, W, S i E) realizuje funkcja **rozdaj():**

```
void rozdaj( int tkarty[])
{const char *walor[13] = {"as", "dwojka",
       "trojka","czworka", "piatka",
       "szostka", "siodemka", "osemka", "dziewiatka",
       "dziesiatka", "walet", "dama", "krol"};
printf("\n\n      N        W        S        E \n");
  for (int i = 1; i <=52; i++)
      {for (int j = 0; j < 52; j++)
         { if (tkarty[j] == i)
             printf("   %c %10s ",
                   kolor[j/13],walor[j%13]);
         }
       if (i%4 == 0) printf("\n");
      }
}    // ----------- koniec rozdaj() --------
```

Należy zwrócić uwagę na fakt, że funkcja **rozdaj()** przetwarza tablicę jednowymiarową! W wywołaniu funkcji:

```
        rozdaj(karty[0]);
```

funkcji **rozdaj()** przekazano argument **karty[0].** Nadaje to wskaźnikowi tkarty wartość równą adresowi elementu **karty [0][0].** W wyniku tego **tkarty[0]** odpowiada elementowi **karty[0][0],** a **tkarty[1]** – elementowi **karty[0][1].** Element **tkarty[13]** odpowiada elementowi **karty[1][0],** czyli pierwszemu elementowi z drugiego wiersza. Inaczej mówiąc, po przyporządkowaniu pierwszego wiersza, przechodzimy do następnego. W ten sposób przyporządkowujemy całą 52 elementową tablicę dwuwymiarową (4 x 12) tablicy jednowymiarowej o 52 elementach.
Należy zauważyć, że implementacje zgodne z K&R pozwalają tak postępować, natomiast kompilatory ANSI C mogą zasygnalizować uwagę, że przekazywany parametr kłóci się z prototypem, który stwierdza, że argument powinien być wskaźnikiem do **int** , a nie wskaźnikiem do wskaźnika. W następującym fragmencie funkcji **rozdaj()**:

```
for (int i = 1; i <=52; i++)
   { for (int j = 0; j < 52; j++)
       { if (tkarty[j] == i)
           printf("   %c %10s ",
               kolor[j/13],walor[j%13]);
       }
   }
```

realizowane jest przeglądanie tablicy **tkarty[],** która w istocie jest tablicą

dwuwymiarową **karty[][].** Wybieramy kolejne karty z talii i rozdzielamy je czterem graczom N, W, S i E. Program do symulowania tasowania i rozdawania kart pokazano na listingu 16.18.

Listing 16.18. Tasowanie talii kart

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

const char kolor[4] = {'\003', '\004', '\005','\006'};
void tasuj  ( int [][13]);
void talia  ( int [][13]);
void rozdaj ( int []);

int main()
{
 int karty [4][13] = { 0 };
 srand (time(0));
 tasuj ( karty );
 talia( karty );
 rozdaj(karty[0]);
 getche();
 return 0;
} //----- koniec int main() ------------

void tasuj ( int tkarty[][13])
{
 int kolor, walor;
 for (int karta = 1; karta <= 52; karta++)
  {
    do {kolor = rand() % 4;
        walor = rand() % 13;
       } while (tkarty[kolor][walor] != 0 );
    tkarty[kolor][walor] = karta;
  }
}   // -- koniec tasuj() --------------

void talia( int tkarty [][13])
{ printf("\n    As    2    3    4    5    6");
  printf("       7    8    9   10    W    D    K\n");
   for (int w = 0; w <=3; w++)
        { printf("\n %c", kolor[w]);
          for (int k = 0; k <= 12; k++)
           printf (" %3d ", tkarty[w][k]);
        }
}    // ---- koniec talia() -------------

void rozdaj( int tkarty[])
```

```
{const char *walor[13] = {"as", "dwojka",
      "trojka","czworka", "piatka",
      "szostka", "siodemka", "osemka", "dziewiatka",
      "dziesiatka", "walet", "dama", "krol"};
 printf("\n\n     N       W       S       E \n");
 for (int i = 1; i <=52; i++)
   {for (int j = 0; j < 52; j++)
       {if (tkarty[j] == i)
          printf("   %c %10s ",
            kolor[j/13],walor[j%13]);
       }
    if (i%4 == 0) printf("\n");
   }
}
```

Po uruchomieniu programu mamy następujący wydruk:

| | As | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | W | D | K |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ♥ | 12 | 51 | 49 | 7 | 22 | 29 | 44 | 14 | 30 | 47 | 46 | 20 | 41 |
| ♦ | 1 | 9 | 37 | 43 | 4 | 42 | 23 | 13 | 10 | 6 | 17 | 40 | 15 |
| ♣ | 19 | 50 | 5 | 45 | 25 | 32 | 34 | 18 | 52 | 21 | 31 | 26 | 16 |
| ♠ | 8 | 28 | 48 | 24 | 38 | 27 | 35 | 11 | 36 | 2 | 3 | 39 | 33 |

| N | W | S | E |
|---|---|---|---|
| ♦ as | ♠ dziesiatka | ♠ walet | ♦ piatka |
| ♣ trojka | ♦ dziesiatka | ♥ czworka | ♠ as |
| ♦ dwojka | ♦ dziewiatka | ♠ osemka | ♥ as |
| ♦ osemka | ♥ osemka | ♦ krol | ♣ krol |
| ♦ walet | ♣ osemka | ♣ as | ♥ dama |
| ♣ dziesiatka | ♥ piatka | ♦ siodemka | ♠ czworka |
| ♣ piatka | ♣ dama | ♠ szostka | ♠ dwojka |
| ♥ szostka | ♥ dziewiatka | ♣ walet | ♣ szostka |
| ♠ krol | ♣ siodemka | ♠ siodemka | ♠ dziewiatka |
| ♦ trojka | ♠ piatka | ♠ dama | ♦ dama |
| ♥ krol | ♦ szostka | ♦ czworka | ♥ siodemka |
| ♣ czworka | ♥ walet | ♥ dziesiatka | ♠ trojka |
| ♥ trojka | ♣ dwojka | ♥ dwojka | ♣ dziewiatka |

Rys. 16.5. Przykładowy wynik uruchomienia programu symulującego tasowanie i rozdawanie kart czterem graczom.

Należy zauważyć, że funkcja wybierająca karty jest nieefektywna, ponieważ za każdym razem przeglądana jest cała tablica.

# BIBLIOGRAFIA


[1] B.W. Kernighan, D.M. Ritchie, Język ANSI C, Wydawnictwo Naukowo-Techniczne, Warszawa, 2003.

[2] C. L. Tondo, S.E. Gimpel, Język ANSI C, ćwiczenia i rozwiązania, Wydawnictwo Naukowo-Techniczne, Warszawa, 2003.

[3] B. Stroustrup, Programowanie, teoria i praktyka z wykorzystaniem C++, Wydawnictwo Helion, Gliwice, 2010.

[4] Kelly, I. Pohl, A book on C, The Benjamin/Cummings Publishing Company, Inc, London, 1985.

[5] R. Jones, I. Stewart, Sztuka programowania w języku C, Wydawnictwo Naukowo-Techniczne, Warszawa, 1992.

[6] C. Delannoy, Ćwiczenia z języka C, Wydawnictwo Naukowo-Techniczne, Warszawa, 1993.

[7] Drozdek, D. L. Simon, Struktury danych w języku C, Wydawnictwo Naukowo-Techniczne, Warszawa, 1996.

[8] S. Prata, Język C, Szkoła programowania, Wydawnictwo Robomatic, Warszawa,1999.

[9] H. Schildt, Programowanie: C, Wydawnictwo Robomatic, Warszawa, 2002.

[10] K.A. Reek, Język C, wskaźniki, Wydawnictwo Helion, Gliwice, 2003.

[11] K. Loudon, Algorytmy w C, Wydawnictwo Helion, Gliwice, 2003.

[12] M.M. Stabrowski, Język C w przykładach, Wydawnictwo Politechniki Lubelskiej, Lublin, 2005.

[13] S. G. Kochan, Język C, wprowadzenie do programowania, Wydawnictwo Helion, Gliwice, 2005.

[14] W.M. Turski, Propedeutyka informatyki, Państwowe Wydawnictwo Naukowe, Warszawa, 1985.

[15] M. Marcotty, H. Ledgard, W kręgu języków programowania, Wydawnictwo Naukowo-Techniczne, Warszawa, 1991.

[16] D. Harel, Rzecz o istocie informatyki, algorytmika, Wydawnictwo Naukowo-Techniczne, Warszawa, 1992.

[17] H. Schildt, Programowanie C++, Wydawnictwo RM, Warszawa, 1998.

[18] P. Van Roy, S. Haridi, Programowanie, koncepcje, techniki i modele, Wydawnictwo Helion, Gliwice, 2005.

[19] Kingsley-Hughes, K. Kingsley-Hughes, Od podstaw programowanie, Wydawnictwo Helion, 2006.

# SŁOWNIK

Angielsko - polski słowniczek języka C

## A

| | |
|---|---|
| address | adres, jest to liczba wskazująca miejsce w pamięci, |
| algorithm | algorytm, opis operacji wykonywanych przez program |
| alignment | ustawienie |
| allocate | alokacja, przydział (pamięci) |
| application | aplikacja, program komputerowy, traktowany przez użytkownika jako jednostka |
| argument | argument, wartość przekazywana do funkcji |
| array | tablica, sekwencja jednakowych obiektów alokowanych w pamięci, w sąsiadujących ze sobą komórkach |
| argument | argument tablicowy, jest to adres pierwszego elementu tablicy |
| declaration | deklaracja tablicy, |
| dimension | rozmiar tablicy, jest to liczba elementów w tablicy, |
| indexing | indeksowanie tablicy, indeks numeruje element tablicy |
| initialization | inicjalizacja, przypisanie wartości elementom tablicy |
| size | rozmiar tablicy, jest to liczba elementów w tablicy, |
| string | tablica znaków, łańcuch w stylu języka C zakończony jest znakiem \0 |
| ASCII | American Standard Code for Information Interchange, kod ASCII – jest to standardowy siedmiobitowy kod wykorzystywany do reprezentowania znaków |
| assertion | asercja, instrukcja mówiąca, że element umieszczony w danym miejscu w programie musi być prawdziwy |
| assignment expression | wyrażenie przypisania, |
| assignment statement | instrukcja przypisania, jest to wyrażenie przypisania zakończone średnikiem |
| auto | automatyczna, nazwa jedna z klas pamięci, |

## B

| | |
|---|---|
| backslash | lewy ukośnik (symbol \ ) |
| binary | binarny (dwójkowy), liczbowy system pozycyjne o podstawie 2 |
| binary operator | operator dwuargumentowy |
| bit | bit, cyfra binarnego system liczbowego (0 i 1) |
| bit field | pole bitowe |

| | |
|---|---|
| bit operators | operatory bitowe, dzięki nim, wykonuje się logiczne operacje bitowe na operandach operatora |
| and | bitowy operator I (symbol &) |
| complement | bitowy operator uzupełnienia (symbol ~) |
| exclusive or | bitowy operator lub wykluczające (symbol ^) |
| inclusive or | bitowy operator lub (symbol \|) |
| shift | bitowy operator przesunięcia (symbol << oraz >>) |
| bitwise | bitowy |
| bitwise operator | operator bitowy |
| boolean | wbudowany typ logiczny, wartości true lub false |
| boolean data | dane typu logicznego (typu Boolean), true (1) oraz false (0) |
| brace | klamra, nawiasy klamrowe (symbol { } ) |
| bracket | nawias kwadratowy (symbol [ ] ) |
| bug | błąd w kodzie, nieoczekiwana sytuacja, termin związany z dr Grace Hopper, która była kontradmirałem w U.S.Navy |
| byte | bajt, osiem bitów |

C

| | |
|---|---|
| call | wywołać, wywołanie |
| call by reference | proces deklarowania parametru funkcji jako zmiennej wskaźnikowej, w konsekwencji przekazanie adresu jako argument |
| call by value | proces deklarowania parametru funkcji jako zmiennej prostej (nie wskaźnikowej), w konsekwencji przekazanie wartości zmiennej jako argument |
| calling function | wywołanie funkcji, przekazanie sterowania do funkcji, która wykona żądane operacje |
| case | przypadek, opcja, słowo kluczowe w konstrukcji switch, etykieta skojarzona z case musi być unikalna |
| case sensitive | rozróżnianie małych i dużych liter w języku C |
| cast | dosłowna, wymuszona, konwersja typów, np. (double)i |
| character | znak, element języka angielskiego, znak ma przypisaną wartość numeryczna typu integer, każdy znak ma przypisaną 7-bitowa reprezentację ASCII |
| code | kod, program lub część programu |
| coding | kodowanie, pisanie program komputerowego |
| collating sequence | porządek sortowania |
| command line | wiersz poleceń |
| comment | komentarz |
| commutative operator | operator przemienny |
| comparison | porównanie |
| compilation | kompilowanie, przekształcanie kodu źródłowego w kod |

| | |
|---|---|
| | maszynowy |
| compiled language | język kompilowany |
| compiler | kompilator, program przekształcający kod źródłowy w kod maszynowy |
| compiler error | błąd kompilacji |
| compile-time errors | błędy wykryte w fazie kompilacji, ( błędy syntaktyczne) |
| compound statement | instrukcja złożona |
| computer program | kombinacja danych i instrukcji przeznaczona do wykonania na komputerze |
| concatenate | połączenie, sklejenie wielu łańcuchów w jeden łańcuch |
| condition | warunek |
| console application | proces tworzenia prostych programów na platformach programistycznych z pominięciem okienek |
| constant | stała (wartość, która nie może sie zmieniać) |
| continue | słowo kluczowe języka C, kontynuacja |
| control expression | wyrażenie sterujące |
| conversion automatic | proces ujednolicania typów w wyrażeniach mieszanych |
| CPU | procesor |

D

| | |
|---|---|
| data types | typy danych |
| debugging | debugowanie, proces usuwania błędów w programach komputerowych |
| decimal numbers | liczba dziesiętna |
| declaration | deklaracja, specyfikowanie typów |
| default | słowo kluczowe (instrukcja switch) |
| define | zdefiniowanie (dyrektywa preprocesora) |
| definition | definicja, jest to deklaracja alokująca pamięć |
| digit | cyfra, zbiór symboli konkretnego system liczbowego |
| double | typ zmiennej, reprezentuje liczbę zmiennoprzecinkową z większą dokładnością niż typ float |
| double precision numbers | liczba podwójnej precyzji, liczba typu double |
| dynamic allocation | dynamiczne przydzielanie pamięci jest procesem zachodzącym w czasie wykonywania program w odróżnieniu od statycznego procesu przydzielania pamięci w fazie kompilacji |

E

| | |
|---|---|
| efficiency | wydajność, wydajność programu oznacza zadawalającą szybkość wykonania, mając do dyspozycji określone zasoby |
| encryption | szyfrowanie |
| end of file | koniec pliku |
| end of string | łańcuch jest zakończony symbolem końca (\0) |

| | |
|---|---|
| enumeration type | typ wyliczeniowy |
| equality expression | wyrażenie relacyjne równości |
| error messages | komunikat błędu |
| errors | błędy |
| compile-time | błędy wykryte w czasie kompilacji |
| logic | błędy logiczne |
| programming | błędy programistyczne |
| run-time | błędy wykryte w czasie wykonywania programu |
| syntax | błędy syntaktyczne |
| typos | błędy typograficzne |
| escape sequence | znaki niedrukowalne (np. \n oznacza skok do nowej linii) |
| evaluate | ocena, wyznaczanie wartości |
| exit | wyjście, najczęściej jest to zakończenie wykonywania programu lub jakiejś jego części |
| executable program | program wykonywalny |
| exponential notation | notacja wykładnicza, jest to sposób zapisywania liczb (np. w notacji wykładniczej zapis 1.625e3 oznacza liczbę dziesiętną 1625, a w notacji naukowej liczbę $1.625x10^3$) |
| exponential part | w reprezentacji liczby zmiennoprzecinkowej jest to część wykładnicza |
| expression | wyrażenie |
| assignment | wyrażenie przypisania |
| bitwise | wyrażenie bitowe |
| comma | wyrażenie z operatorem przecinkowym |
| constant | wyrażenie stałe, |
| equality | wyrażenie równości |
| logical | wyrażenie logiczne |
| relational | wyrażenie relacyjne |
| extern | typ zewnętrznej klasy pamięci, gdy zmienna jest deklarowana poza funkcją (zmienna globalna), przydziela sie jej pamięć klasy extern |
| external storage class | zewnętrzna klasa pamięci |

F

| | |
|---|---|
| factorial | silnia (symbol n!) jest to funkcja rekurencyjna |
| false | fałsz, zmienna typu bool |
| fibonacci numbers | liczby Fibbonacci'ego, sekwencja liczb zdefiniowanych rekurencyjnie |
| field | pole, składnik struktury |
| file | plik, jest to obszar, w którym przechowywane są dane |
| FIFO | first in, first out, pierwszy na wejściu, pierwszy na wyjściu |

| | |
|---|---|
| file extension | rozszerzenie pliku |
| float | zmiennoprzecinkowy typ danych prostych |
| floating constants | stała zmiennoprzecinkowa |
| floating types | typy zmiennoprzecinkowe |
| flow of control | przebieg sterowania |
| flush | opróżnić (bufor) |
| for statement | instrukcja pętli for |
| formal parameter | parametr formalny |
| format | specyfikatory konwersji (np. w funkcji printf() format %f) |
| fractional part | część ułamkowa, w zapisie liczby zmiennoprzecinkowej |
| function | funkcja, nazwany, wydzielony fragment programu, logiczna jednostka przetwarzania |
| function | funkcja |
| argument list | lista parametrów (argumentów) |
| arguments structure | parametry typu struktury przekazywane do funkcji |
| array arguments | argument tablicowe |
| body | ciało funkcji |
| call by reference | wywołanie funkcji z argumentem typu wskaźnikowego |
| call by value | wywołanie funkcji z argumentem prostym (nie wskaźnikowym) |
| definition | definicja funkcji |
| formal parameter | parametry formalne |
| header | nagłówek funkcji |
| parameter | parametr funkcji |
| pointer parameter | parametr typu wskaźnikowego przekazywany do funkcji |
| recursion | funkcja rekurencyjna |
| type specifier | specyfikator typu zwracanej wartości funkcji |

G

| | |
|---|---|
| global | globalny |
| goto statement | instrukcja goto |

H

| | |
|---|---|
| header file | plik nagłówkowy (dołączany do program) |
| heap | stos, sterta (typ pamięci) |
| hexadecimal | szesnastkowy (system liczbowy) |
| hexadecimal digit | cyfra systemu (liczbowego) szesnastkowego |
| high level languages | język programowania wysokiego poziomu |
| high-order bit | bit najbardziej znaczący |

I

| | |
|---|---|
| IDE | Integrated Development Environment, zintegrowane środowisko programistyczne |
| identifiers | identyfikator, nazwa |
| if statement | instrukcja if |
| implementation | implementacja, pisanie i testowanie program, także – kod programu |
| index | indeks |
| include | dołączyć |
| include file | plik dołączany |
| infinite loop | pętla nieskończona |
| initialization | inicjalizacja |
| array | inicjalizacja tablicy |
| pointer | inicjalizacja wskaźnika |
| string | inicjalizacja łańcucha |
| structure | inicjalizacja struktury |
| variable | inicjalizacja zmiennej |
| input/output | wejście/wyjście, dane wejściowe są wprowadzane do programu, dany wyjściowe są wytwarzanie w procesie przetwarzania |
| integer | całkowity, typ zmiennej |
| integer overflow | przepełnienie, w czasie wykonania program może wystąpić przekroczenie zakresu np. dla liczb całkowitych, można otrzymać niepoprawny wynik |
| integer part | część całkowita , w zapisie liczb zmiennoprzecinkowych |
| integer types | całkowity typ zmiennych |
| interface | interfejs, zbiór deklaracji, potrzebnych do wywoływania funkcji |
| invariant | niezmiennik, element, który w danym punkcie programu musi być prawdziwy |
| item | pozycja, składnik,egzemplarz |
| iteration | iteracja, wielokrotne wykonywanie tych samych instrukcji |

K

| | |
|---|---|
| Keyword | słowo kluczowe (zastrzeżone) |

L

| | |
|---|---|
| label | etykieta (w instrukcji switch oraz goto) |
| labeled statement | etykieta instrukcji ( wymagana przy skoku goto) |
| languages | języki programowania |
| high-level | języki programowania wysokiego poziomu |
| low-level | języki programowania niskiego poziomu |
| middle-level | języki programowania średniego poziomu |
| object-oriented | języki programowania zorientowane obiektowo |

| | |
|---|---|
| procedure-oriented | proceduralne języki programowania |
| library | biblioteka, kolekcja plików, pliki zawierają typy, funkcje, struktury, dyrektywy, itp., które są wykorzystywane w wielu programach |
| LIFO | last in, first out, ostatni na wejściu, pierwszy na wyjściu |
| linker | konsolidator, program łączący, jest to program który konsoliduje inne pliki i biblioteki z programem obiektowym, w ten sposób otrzymujemy program wykonywalny |
| literal | literał, bezpośredni zapis wartości, np. 13 interpretujemy jako wartość „trzynaście” |
| local | lokalny |
| logical expression | wyrażenie logiczne |
| logical operators | operatory logiczne |
| long | typ danych |
| loop | pętla, sekwencja instrukcji, która jest wykonywana wielokrotnie |
| loops | pętle |
| for | pętle for |
| infinite | pętle nieskończone (wykonujące sie bez końca) |
| inner | pętle wewnętrzne |
| nested | pętle zagnieżdżone |
| outer | pętle zewnętrzne |
| while | pętle while |
| low-level | niskiego poziomu |
| low-order bit | bit najmniej znaczący |
| lvalue | l-wartość, |

M

| | |
|---|---|
| machine accuracy | dokładność maszyny liczącej, w metodach numerycznych wygodnie jest zadeklarować eps („epsilon maszynowe”) jako najmniejszą dodatnią wartość zmiennej typu double spełniającą warunek, że wyrażenie 1.0 < 1.0+eps jest prawdziwe |
| machine dependent | zależny od maszyny liczącej |
| machine language | język maszynowy ( wewnętrzny język komputerowy, składający się z serii zer i jedynek) |
| macro argument | argument przekazywany do makrodefinicji |
| macro | makrodefinicji (zdefiniowany skrót, konstrukcje preprocesora) |
| magic numbers | liczby magiczne, tak w żargonie komputerowym nazywane są liczby pojawiające się w indywidualnych instrukcjach |
| main() | słowo kluczowe, zastrzeżona nazwa dla funkcji, od |

| | |
|---|---|
| | której zaczyna sie wykonywanie programu |
| mask | maska, jest to stała lub zmienna, która jest wykorzystywana do wyodrębnienia żądanego bitu w zmiennej, używana w operacjach bitowych |
| mathematical library | funkcje matematyczne zebrane w pliku (np. <math.h> |
| member | element struktury, składnik struktury |
| mixed mode expressions | arytmetyczne wyrażenie, które zawiera całkowite i niecałkowite operandy (np. dodawanie zmiennych typu int i float) |
| modulus operator | operator reszty (modulo, symbol operator %), np. 9 % 4 = 1 |
| multi-dimensional arrays | wielowymiarowa tablica |

N

| | |
|---|---|
| nested loop | zagnieżdżona pętla |
| newline | znak nowej linii |
| nibble | półbajt, 4 bity, amerykański termin |
| nonprinting character | znak niedrukowalny, znaki ASCII nie wyświetlane na ekranie, symbolizują czynności takie jak np. cofanie, tabulacja, przejście do nowej linii |
| not operator | operator logiczny nie (symbol !) |
| null character | znak zera (symbol \0), używany np. do oznaczania końca łańcucha |
| null pointer value | wskaźnik zerowy, wskaźnik o wartości 0 i NULL nie wskazuje żadnej wartości |

O

| | |
|---|---|
| object program | program pośredni, jest to kod wyprodukowany z kodu źródłowego przez kompilator, potrzebny jest jeszcze konsolidator aby wyprodukować kod wykonywalny |
| octal digit | cyfra ósemkowa |
| octal number | liczba ósemkowa |
| one-dimensional array | tablica jednowymiarowa |
| one's complement | dopełnienie jedynkowe lub bitowa negacja (symbol ~), jednoargumentowy operator ~ zmienia każde zero na jedynkę, a każdą jedynkę na zero |
| operating system | system operacyjny |
| operator | operator |
| addition | operator dodawania |
| address | operator adresowy |
| arithmetic | operator arytmetyczny |
| assignment | operator przypisania |
| associativity | łączenie operatora (z lewej do prawej lub z prawej do lewej) |

bitwise operator bitowy
bitwise exclusive or operator alternatywy bitowej (symbol |)
cast (type) operator rzutowania jawnego
comma operator przecinkowy
conditional operator warunkowy (symbol (?:)
decrement operator dekrementacji
dereferencing operator dereferencji (wyłuskania)
division operator dzielenia
equality operator relacyjny (porównywania)
equals operator równości
greater than operator relacyjny większy niż
greater than or equal to operator relacyjny większy niż lub równy
increment operator inkrementacji
left shift operator przesunięcia bitowego
less than operator relacyjny mniejszy niż
less than or equal to operator relacyjny mniejszy niż lub równy
logical operator logiczny
logical and operator logiczny koniunkcji (symbol &&)
logical negation operator logiczny zaprzeczenia
logical or operator logiczny alternatywy (symbol ||)
modulus operator modulo (symbol %)
multiplication operator mnożenia
not equals operator relacyjny nierówny
one's complement operator bitowy dopełnienia
precedence priorytet operatora (określa kolejność wykonywania operacji)
relational operator relacyjny
right shift operator bitowy przesunięcia w prawo
sizeof operator rozmiaru
structure member operator dostępu do elementu struktury (symbol . )
structure pointer operator dostępu do element struktury przez wykorzystanie wskaźnika (symbol ->)
subtraction operator odejmowania
unary minus jednoargumentowy operator minus
output wyjście
overflow przepełnienie (kategoria błędu)

P

parameter parametr, argument
parantheses nawias okrągły (symbol ( ) )
pass przekazać (np. argumenty)
pointer wskaźnik
argument argument wskaźnikowy (np. przekazywanie wartości do funkcji)

| | |
|---|---|
| arithmetic | arytmetyka wskaźnikowa, specjalny sposób manipulowania wskaźnikami (np. inkrementacja) |
| array | tablica wskaźników |
| assignment | przypisanie wskaźnika (nadawanie wartości zmiennej wskaźnikowej) |
| constant | stała wskaźnikowa |
| declaration | deklaracja wskaźnika |
| indirection | wskazanie pośrednie, dereferencja zmiennej wskaźnikowej |
| initialization | inicjalizacja wskaźnika |
| parameter | parametr wskaźnikowy |
| polish expression evaluator | obliczanie wyrażenia zapisanego przy pomocy polskiej notacji (wyrażenie bez użycia nawiasów) |
| polish (parenthesis-free) notation | notacja polska, sposób zapisywania wyrażeń, np. wyrażenie 3 + 9, w polskiej notacji ma postać 3, 9, + |
| pop | zdjąć (np. ze stosu) |
| position notation | zapis pozycyjny (w systemie liczbowym) |
| postfix operator | operator postfiksowy inkrementacji lub dekrementacji (np. x++) |
| precedence | priorytet, kolejność wykonania sekwencji operatorów, |
| precision | dokładność, termin ten oznacza ilość znaczących miejsc dziesiętnych po kropce w zapisie liczby zmiennoprzecinkowej |
| prefix operator | operator prefiksowy inkrementacji lub dekrementacji (np. ++x) |
| preprocessor | preprocessor, specjalny program przetwarzający wstępnie kod programu języka C |
| prime number | liczba pierwsza |
| print | drukować, wyświetlać |
| program flow | przebieg programu |
| programming | programowania |
| programming language | język wykorzystywany w pisaniu programów |
| promote | awansować (typ) |
| pseudo random number generator | generator liczb pseudolosowych |
| push | położyć (na stosie) |

R

| | |
|---|---|
| random number generator | generator liczb losowych |
| range | zakres, określa największą i najmniejszą dodatnią zmiennoprzecinkową liczbę, jaka może być |

| | |
|---|---|
| | reprezentowana w konkretnym komputerze |
| real number | liczba rzeczywista |
| recursion | rekurencja, proces występuje, gdy funkcja wywołuje sama siebie |
| register | słowo kluczowe, klasa pamięci register ( rejestr) |
| relational expression | wyrażenie relacyjne |
| remainder | reszta z dzielenia |
| remainder operator | operator reszty, operator modulo (symbol %) |
| reserved word (keyword) | słowo zastrzeżone (kluczowe) |
| return statement | instrukcja return |
| return value | wartość zwracana |
| round off error | błąd obcinania |
| run-time | czas wykonania (czas wykonywania program) |
| run-time terror | błąd wykonania programu |
| rvalue | r-wartość |

S

| | |
|---|---|
| scalar dot product | iloczyn skalarny (dla wektorów) |
| scope | zakres widoczności (zmiennych) |
| separator | separator (przecinek spacja, itp.) |
| sequential access | dostęp sekwencyjny |
| shift | przesunięcie |
| short | słowo kluczowe dla oznaczenia typu zmiennej |
| side effect | efekt uboczny |
| signed | ze znakiem, specyfikator typu |
| significant figures | cyfry znaczące (zapis liczb zmiennoprzecinkowych) |
| sizeof() | operator rozmiaru, zwraca rozmiar pamięci, jaki zajmuje zmienna lub dany typ (w bajtach) |
| sorting | sortowanie |
| bubble sort | sortowanie bąbelkowe |
| efficiency | wydajność sortowania |
| heapsort | sortowanie stogowe |
| mergesort | sortowanie typu mergesort |
| quicksort | sortowanie typu quicksort |
| source code | kod źródłowy |
| source file | plik źródłowy |
| sqrt() | pierwiastek kwadratowy z dodatniej liczby, biblioteka math.h |
| stack | stos |
| stack operation | operacja na stercie (typ pamięci) |
| standard library | biblioteka standardowa |
| statement | instrukcja |
| assignment | instrukcja przypisania |
| break | instrukcja break (przerwania) |

| | |
|---|---|
| switch statement | instrukcja switch (wyboru) |
| syntactic symbol | symbol syntaktyczny |
| syntax | syntaks |
| syntax error | błąd syntaktyczny |

T

| | |
|---|---|
| tab character | znak tabulacji (symbol \t) |
| text editor | edytor tekstowy |
| time | czas |
| token | tokenem języka komputerowego jest każda sekwencja znaków, która ma unikalne znaczenie |
| top down | metoda "top down" jest jedną z technik programowania |
| true | logiczny typ zmiennej (bool) |
| truncate | obciąć (cyfry znaczące) |
| truth table | tablica, która pokazuje wynik działania funkcji logicznych (bool) |
| twin prime | liczby pierwsze bliźniacze, jeżeli p i p+2 są pierwsze, to są bliźniacze |
| two-dimensional array | tablica dwuwymiarowa |
| two's complement | uzupełnienie do dwóch, jest to jeden ze sposobów reprezentacji liczb całkowitych |
| type enum | typ wyliczeniowy |
| type int | typ całkowity |
| type mismatch | niezgodność typów |
| type specifier | specyfikator typu |
| typedef | słowo kluczowe, z jego pomocą definiujemy własny typ |

U

| | |
|---|---|
| unary operator | operator jednoargumentowy |
| union | unia, struktura danych |
| unsigned | bez znaku, specyfikator typu, |

V

| | |
|---|---|
| variable | zmienna |
| void | typ danych, słowo kluczowe, |

W

| | |
|---|---|
| while statement | instrukcja while (pętlia) |
| white space | znaki niedrukowalne (np. spacja) |
| word | słowo (sposób organizacji bitów), podstawowa jednostka pamięci w komputerze |

# SKOROWIDZ