Jerzy Kisilewicz

Język C

w środowisku Borland C++

Wydanie IV

Oficyna Wydawnicza Politechniki Wrocławskiej
Wrocław 2003

Opiniodawca
Marian ADAMSKI

Opracowanie redakcyjne i korekta
Hanna BASAROWA

Projekt okładki
Dariusz Godlewski

OFICYNA WYDAWNICZA POLITECHNIKI WROCŁAWSKIEJ, Wrocław 1998
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

ISBN 83-7085-327-7

Drukarnia Oficyny Wydawniczej Politechniki Wrocławskiej.

Nakład + egzemplarzy
Ark.: wyd. , druk. .

Spis treści

1. Wprowadzenie

Język C został opracowany w 1972 roku w Bell Laboratories. Jego autorzy B.W. Kernighan i D.M. Ritchie, opisali wzorcową wersję tego języka w książce pt. „Język C", a następnie, po powstaniu normy ANSI, w książce „Język ANSI C". Język C kontynuuje linię wyznaczoną przez takie języki, jak Algol i Pascal. Dziedziczy po nich podstawowe pojęcia i zasady budowy instrukcji, wyrażeń i programów, jak np.: budowanie programów z funkcji, blokową budowę funkcji, zasięg działania identyfikatorów, zasady przykrywania identyfikatorów o identycznych nazwach, zasady przekazywania zmiennych do funkcji oraz budowę pętli.

Język C doczekał się wielu zastosowań, z których najpopularniejszym jest Borland C++. Wersja 2.0 języka Borland C++ została dobrze opisana przez Jana Bieleckiego w książkach „Borland C++. Programowanie strukturalne", „Borland C++. Programowanie obiektowe" i „Borland C++. Biblioteki standardowe". Jedną z bardziej popularnych wersji językowych do pisania programów zarówno obiektowych, jak i nieobiektowych uruchamianych pod systemem DOS jest wersja 3.1. Kolorowy edytor tej wersji znakomicie ułatwia pisanie programów i wykrywanie błędów. Do pisania programów uruchamianych w 32-bitowym środowisku Windows'95 lub Windows NT doskonale nadaje się wersja 4.5 kompilatora Borland C++.

1.1. Porównanie z językiem Pascal

Programy wielu szkół zakładają nauczanie języka C po opanowaniu języka Pascal. Podczas pisania tej książki przyjęto, że czytelnik poznał wcześniej język Pascal oraz będzie początkowo uruchamiał swoje programy nieobiektowym kompilatorem platformy Borland C++. Platforma ta zawiera bowiem kompilator obiektowy i nieobiektowy. Aby używać kompilatora nieobiektowego, należy plikom źródłowym

nadać rozszerzenie *.C i kompilować z ustawioną opcją *Options | Compiler | C++ options | Use C++ Compiler | CPP extension*. Opisano tylko niektóre funkcje zakładając, że czytelnik wykorzystując je oraz kontekstowy „help" łatwo zapozna się ze wszystkimi potrzebnymi mu w danej chwili i oferowanymi przez kompilator funkcjami.

Dobre opanowanie języka Pascal bardzo ułatwia naukę języka C, gdyż oba te języki mają wiele cech wspólnych. Cechy te są następujące:

- Programy zbudowane są z funkcji, które wywołując się, przekazują między sobą dane za pomocą argumentów oraz zmiennych globalnych, a wyniki mogą zwracać przez nazwę funkcji. Funkcje mogą zmieniać wartości dostępnych im zmiennych globalnych oraz innych wskazanych zmiennych zewnętrznych. Parametry przekazywane przez wartość są kopiowane do lokalnych zmiennych funkcji.
- Każda funkcja jest blokiem zamkniętym w nawiasy { } (begin–end), zawierającym definicje, deklaracje i instrukcje.
- Każdy ciąg instrukcji można zamknąć w nawiasy { }, tworząc instrukcję zblokowaną, która jest semantycznie równoważna instrukcji prostej i może wystąpić tam, gdzie opis języka wymaga pojedynczej instrukcji.
- Definiowane i deklarowane zmienne obejmują swoim zasięgiem obszar od miejsca definicji lub deklaracji do końca swego bloku, włączając w to wszystkie bloki wewnętrzne, w których nie zdefiniowano lub zadeklarowano innej zmiennej o tej samej nazwie. Zmienne zadeklarowane w bloku przesłaniają zmienne o tych samych nazwach zadeklarowane na zewnątrz.
- Instrukcje *if*, *switch* oraz instrukcje pętli są skonstruowane podobnie, choć ich działanie może być nieco odmienne.

O ile Pascal jest uporządkowany zgodnie z regułami matematyki, to język C ma prostszą logikę do poziomu techniki, w której wszystkie typy danych są pamiętane jako liczby lub ciągi liczb. Dzięki temu jest on bardziej „elegancki" i daje większą swobodę programowania. Tak np. można dodawać znaki do liczb rzeczywistych, podobnie jak liczby do wartości logicznych. Wszystko bowiem, co jest fałszem. jest równe zeru, a co jest prawdą, jest równe jedności. Na uwagę zasługują następujące uproszczenia:

- Wszystkie wartości skalarne (np. znakowe, logiczne) są liczbowe (całkowite lub zmiennoprzecinkowe). Dozwolone są zatem wyrażenia takie jak $'A'+1$ czy $2+(x>y)$.
- Wywołanie funkcji zawsze zawiera nawiasy ( ), nawet gdy funkcja nie ma parametrów, np.: *c*=*getch*( ); lub *clrscr*( );.
- Nie wprowadza się pojęcia procedury. Rolę jej pełni funkcja, która nie zwraca wyniku przez swoją nazwę. Poprawne są więc instrukcje: $y=sin(x)$; oraz *sin*(*x*);.
- Parametry funkcji przekazywane są tylko przez wartość, chociaż w języku C++ również przez referencję czyli przez zmienną.

- Średnik nie jest separatorem, lecz umieszczony na końcu wyrażenia tworzy z niego instrukcję. Każda instrukcja kończy się więc średnikiem (nawet ta przed *else*). Poprawne są też instrukcje $y=sin(x)$; $sin(x)$; $a+b$;.
- Przypisanie jest operatorem i podobnie jak dodawanie i wszystkie inne operatory zwraca określony wynik. Wyrażenie. $x=y$ np ma swoją wartość podobnie jak $x+y$. Dozwolone są też wielokrotne podstawienia, np. $x=y=z$; podobnie jak wielokrotne dodawania np. $x+y+z$;.
- Tablice są jednoindeksowe. Nawiasy [ ] są też operatorem, podobnie jak np. dodawanie i podobnie jak w przypadku x+$y+z$ można tworzyć konstrukcje $A[i][j]$, o ile wynikiem wyrażenia $A[i]$ (elementem tablicy A) jest tablica innych elementów.
- Wszystkie pętle są typu *while*, to znaczy wykonują się tak długo, jak długo jest spełniony zadany warunek.

Podobnie jak w języku Pascal nie ma obowiązku pisania programów w określonej szacie graficznej W publikacjach zaleca się, aby każda instrukcja była napisana w oddzielnym wierszu – tak samo jak w języku Pascal. W tej książce przykładowe programy często przedstawiono w bardziej zwartej formie. Pisanie programów w nieco bardziej zwartej formie niż forma publikacyjna, ale z zachowaniem istotnych wcięć, umożliwia oglądanie na ekranie większych fragmentów kodu źródłowego. To z kolei ułatwia uruchamianie programu.

Przepisanie programów w postaci zalecanej w publikacjach może być traktowane jako dodatkowe ćwiczenie.

1.2. Kompilator i jego środowisko

Platforma programistyczna Borland C++, podobnie jak platforma Borland Pascal, dostarcza programy narzędziowe do przygotowywania, kompilowania, uruchamiania i wykonywania programów. Platforma zawiera: rozbudowany wielookienkowy edytor, kompilator języka C, kompilator obiektowy C++, debugger. Jest ona wyposażona w „kontekstowy help” oraz umożliwia posługiwanie się myszką. Od użytkownika wymaga tego, aby nie dokonywał on żadnych zapisów do katalogu kompilatora ani do żadnego z jego podkatalogów, dlatego też nie powinno się uruchamiać platformy z jej podkatalogu *BIN*. Platformę uruchamia się programem *bc.exe*, który jest umieszczony w podkatalogu *BIN* kompilatora. W momencie uruchomienia program *bc.exe* zakłada w aktualnym katalogu plik tymczasowy, tak więc platformy nie da się uruchomić, jeśli nie posiada się praw zapisu do aktualnego katalogu.

1.2.1. Przygotowanie środowiska

W laboratorium wygodnie jest posługiwać się własną konfiguracją kompilatora i edytora i należy zapewnić sobie przechowanie tej konfiguracji. W tym celu przed pierwszym uruchomieniem programu *bc.exe* należy:

- Przekopiować z podkatalogu BIN katalogu BORLAND C++ do własnego katalogu pliki konfiguracyjne TCDEF.DPR, TCDEF.DSK i TCCONFIG.TC. Powyższe pliki są poszukiwane przez program *bc.exe* najpierw w aktualnym katalogu, z którego ten program został uruchomiony, a następnie są poszukiwane w podkatalogu BIN, w którym program *bc.exe* jest zainstalowany.
- Uruchomić program *bc.exe*, wydając komendę z własnego katalogu, aby użył on prywatnych kopii plików konfiguracyjnych.
- Ustawić pożądaną konfigurację opcji edytora i kompilatora. W szczególności należy wpisać prawidłowe ścieżki w *Options* | *Directories*, włączyć wszystkie ostrzeżenia kompilatora w *Options* | *Compiler* | *Messages* | *Display*, zależnie od wyboru kompilatora wpisać domyślne rozszerzenia nazw edytowanych plików (*C* lub *CPP*) w *Options* | *Environment* | *Editor*.
- Nagrać ustawioną konfigurację i zakończyć edycję komendą Alt-X.

W momencie startu program *bc.exe* zakłada w aktualnym katalogu plik „swapowy", który następnie jest kasowany podczas prawidłowego wyjścia z tego programu. Próba uruchomienia *bc.exe* z katalogu, w którym nie ma się praw zapisu (zwykle z katalogu kompilatora w sieci NOVELL) zakończy się niepowodzeniem, ponieważ nie można w tym katalogu utworzyć pliku.

1.2.2. Edytor

Edytor Borland C++ umożliwia równoczesną edycję wielu plików, z których każdy znajduje się w odrębnym oknie. Edycji podlega plik znajdujący się w aktywnym oknie. Aktywne okno można wybrać wciskając jego numer równocześnie z klawiszem Alt. **Zbędne okna należy zamykać**, ponieważ nadmierna liczba otwartych okien niepotrzebnie zajmuje pamięć operacyjną, spowalniając pracę kompilatora. Domyślnym rozszerzeniem plików jest **.C** lub **.CPP**. W opcjach edytora można to rozszerzenie zmienić.

Przykładowe ważniejsze komendy edytora to:

NEW – otwarcie (kreowanie) nowego dokumentu o tymczasowej nazwie NONAMEnn.C (*nn* – kolejny numer). Nazwa ta będzie mogła być zmieniona podczas zapisywania dokumentu na dysk,

F3 (OPEN) – otwarcie okna i pliku do edycji,

F2 (SAVE) – zapisanie pliku z aktywnego okna na dysk,
Alt-F3 – zamknięcie aktywnego okna,
F1 – otwarcie okna pomocy (help),
Ctrl-F1 – pomoc (help) o słowie na pozycji kursora,
Alt-F5 – pokazanie ekranu wyjściowego („user screen”),
Ctrl-Q[– odszukanie do przodu ogranicznika tworzącego parę z danym,
Ctrl-Q] – odszukanie do tyłu ogranicznika tworzącego parę z danym.

Dwie ostatnie komendy Ctrl-Q[oraz Ctrl-Q] są pomocne do wyszukiwania opuszczonych lub zbędnych nawiasów oraz do analizy struktury blokowej programu. W chwili wydawania komendy kursor musi znajdować się na danym nawiasie (ograniczniku) { } ( ) [ ] < > /* */ " lub ' i przeskakuje do nawiasu będącego z nim w parze. Jeśli takiego nawiasu nie ma, pozycja kursora nie zmienia się.

1.2.3. Kompilator

Aby mieć możliwość analizowania ostrzeżeń kompilatora, proces kompilacji (przynajmniej na początku) dobrze jest podzielić na dwa etapy: kompilację do pliku *.*obj* (*Compile* | *Compile*) oraz łączenie (*Compile* | *Link*). Ostrzeżenia są często wynikiem błędów i powinny zostać usunięte. Na przykład użycie w warunku instrukcji *if* podstawienia ($x=y$) zamiast porównania ($x==y$) wygeneruje ostrzeżenie, nie zaś komunikat błędu.

Przykładowe ważniejsze komendy kompilatora to:

Alt-CC (*COMPILE*) – kompilacja modułu znajdującego się w aktywnym oknie edycyjnym i utworzenie półskompilowanego pliku z rozszerzeniem *.*obj*,
F9 (*MAKE* Alt-CM) – kompilacja modułu znajdującego się w aktywnym oknie edycyjnym lub jeśli jest otwarty projekt, to kompilacja plików wymienionych w projekcie i utworzenie pliku z rozszerzeniem *.*exe*. W każdym przypadku kompilacji podlegają tylko te moduły, których pliki źródłowe nie mają aktualnego pliku *.*obj*,
Alt-C B (*BUILD*) – podobnie jak *MAKE* lecz kompiluje się wszystkie moduły. Przebudowanie programu stosuje się po zmianie opcji kompilatora,
Ctrl-F9 (*RUN*) – wykonanie programu z ewentualną kompilacją jak w przypadku komendy *MAKE*,
Alt-F5 – pokazanie ekranu wyjściowego („user screen”).

1.2.4. Debugger

W środowisko edytora i kompilatora Borland C++ zawiera moduł uruchomieniowy nazywany z angielska debuggerem, który jest pomocny w znajdywaniu błędów wykonania programu. Debugger umożliwia:

- śledzenie przebiegu wykonywania programu,
- analizowanie wyników w miarę ich powstawania,
- obserwowanie zmian wartości zmiennych i wyrażeń,
- przypisanie wybranym zmiennym testowych wartości w czasie wykonywania się programu.

Aby w czasie wykonywania się programu istniała możliwość symbolicznego odwoływania się do obiektów programu źródłowego, należy ustawić opcję debuggera *Options* | *Debugger* | *Source Debugging On*.

Przykładowymi komendami debuggera są:

F4 – wykonanie programu do pozycji kursora lub do pułapki,
F7 – wykonanie instrukcji w wyróżnionym wierszu programu z wchodzeniem do wywoływanych funkcji,
F8 – jak F7, ale bez śledzenia wykonania funkcji.
Ctrl-F4 – uaktywnienie okienka dialogowego *Evaluate and Modify*,
Alt-F4 – pokazanie wartości elementu wskazywanego przez kursor,
Ctrl-F8 – wstawienie/usunięcie pułapki na instrukcji wskazywanej aktualną pozycją kursora w edytorze,
Ctrl-F7 – umieszczenie wyrażenia w okienku obserwacyjnym,
Ctrl-F2 – zakończenie uruchamiania.

Program można zatrzymać przed linią, w której znajduje się kursor, jeśli uruchamia ten program (lub wznawia jego pracę po zatrzymaniu) klawiszem F4. Używając klawiszy F7 lub F8 można wykonywać program linia po linii, wchodząc lub nie do wywoływanych funkcji. Jeśli program został zatrzymany, to można odczytać wartości wybranych zmiennych w oknie *Watch* lub w oknie otwieranym za pomocą klawiszy Ctrl-F4 lub Alt-F4.

Typowymi błędami początkującego programisty w języku C są:

- Blokowanie lub ignorowanie ostrzeżeń kompilatora.
- Pobieżne testowanie programu. Otrzymanie dobrych wyników dla przykładowych danych nie dowodzi poprawności programu.
- Nadużywanie klawiszy Ctrl-F9 lub F9. Istotne ostrzeżenia kompilatora mogą pozostać niezauważone, gdy kompilację, łączenie i wykonanie inicjuje się jedną komendą Ctrl-F9.

1.3. Przykłady prostych programów

Przykład 1.1. Tabela znaków o zadanych kodach

```
#include <conio.h>
void main(void)
  { int z,j,zp=128, zk=256;
    clrscr();
    cprintf("Znaki o kodach od %d do %d", zp, zk-1);
    for(z=zp, j=0; z<zk; z++, j++)
      { gotoxy(1+10*(j/16), 5+j%16);
        cprintf("%d %c", z, z);
      }
    getch();
  }
```

Kolejne wiersze programu mają następujące znaczenie.

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

Instrukcje rozpoczynające się znakiem # są instrukcjami preprocesora i sterują przebiegiem kompilacji. Instrukcja *#include* poleca w jej miejsce dołączyć do programu wymieniony plik. W tym przypadku dołączany jest plik *conio.h*, zwykle z podkatalogu INCLUDE. Plik *conio.h* zawiera definicje i deklaracje konieczne do użycia w programie funkcji obsługi ekranu, takich jak: *clrscr, cprintf, gotoxy* i *getch*. W opcjach *Directories* | *Include* podaje się ścieżkę do włączanych plików, których nazwy ujęto w nawiasy < >, nie zaś w cudzysłowy. Jeśli zachodzi potrzeba włączenia pliku z innego katalogu należy nazwę tego pliku poprzedzoną ścieżką ująć nie w nawiasy < >, lecz w podwójne cudzysłowy.

```
void main(void)
```

Każdy program musi zawierać funkcję główną o nazwie *main*, ponieważ od tej funkcji rozpoczyna się wykonywanie programu. Funkcja *main* może zwracać wartość całkowitą. W tym przypadku funkcja *main* posiada typ *void*, to znaczy, że nie zwraca ona żadnej wartości. Lista argumentów tej funkcji jest tu typu *void*, co oznacza, że funkcja ta nie ma parametrów.

```
  { int z,j,zp=128, zk=256;
```

Nawias klamrowy { (nawias *begin*) otwiera blok instrukcji funkcji. Pierwszą instrukcją jest definicja dwu zmiennych całkowitych *z*, *j* typu *int*. Średnik kończy każdą instrukcję.

```
clrscr( );
cprintf("Znaki o kodach od %d do %d", zp, zk-1);
```

Funkcja *clrscr* czyści ekran i ustawia kursor w jego lewym górnym rogu, czyli na pozycji (1,1). Ekran tekstowy zwykle ma 80 kolumn i 25 wierszy. Tak więc prawy dolny róg ekranu ma pozycję (80,25). Poczynając od pozycji kursora (1,1) funkcja *cprintf* wyprowadza na ekran podany tekst, wstawiając w pola konwersji *%d* liczby *zp* oraz *zk*–1. Pierwszym argumentem funkcji *cprintf* jest format wyprowadzanego tekstu. Jest to wzór tekstu, w którym pola zmienne oznaczono symbolami zaczynającymi się od znaku % (*%d*). Pola te nazywają się polami konwersji. Za znakiem % podany jest kod konwersji, który określa jak interpretować i jak wyprowadzać związany z tym polem kolejny argument funkcji *cprintf*. W przykładzie użyto kodu *d*, który oznacza przedstawienie liczby całkowitej w postaci dziesiętnej. Pola zmienne są wypełniane kolejnymi wartościami, podanymi jako kolejne argumenty za formatem. Tych wartości musi być dokładnie tyle, ile jest pól konwersji. W tym przypadku są dwa pola konwersji dla dwóch liczb: *zp* oraz *zk*.

```
for(z=zp, j=0; z<zk; z++, j++)
```

Instrukcja *for* jest instrukcją pętli. Przed wejściem do pętli podstawia się $z=zp$ oraz $j=0$. Pętla ta wykonuje się tak długo, jak długo jest spełniony warunek $z<zk$. Po każdej iteracji, czyli po każdorazowym wykonaniu wewnętrznej instrukcji pętli, będą wykonywane instrukcje *z*++ oraz *j*++, nakazujące zwiększanie wartości zmiennych z oraz *j* o 1. Tak więc pętla wykonywana jest dla $z=zp,\ zp+1,\ ...,\ zk-1$. Zauważmy, że dla $z=zk$ pętla nie będzie wykonana. Zmienna *z* oznacza tu kod znaku, natomiast *j* jest kolejnym numerem znaku, służącym do określenia położenia wydruku tego znaku na ekranie.

W języku C są operacje inkrementacji ++ oraz dekrementacji --. Podobnie jak *j*++ oznacza zwiększenie *j* o 1, to *j*-- oznacza zmniejszenie *j* o 1. Jeżeli zachodzi potrzeba zmiany wartości zmiennej o więcej niż 1, to można użyć operacji *j*+=*n* lub *j*-=*n*, które polecają odpowiednio zwiększyć lub zmniejszyć *j* o *n* . Analogicznie operacje *j**=*n* lub *j*/=*n* polecają odpowiednio pomnożyć lub podzielić *j* przez *n* .

```
{ gotoxy(1+10*(j/16), 5+j%16);
  cprintf("%d %c", z, z);
}
```

Nawiasy klamrowe { } tworzą zblokowaną instrukcję wewnętrzną pętli *for*, w skład której wchodzą instrukcje wywołania funkcji *gotoxy* i *cprintf*. Funkcja *gotoxy* ustawia pozycję kursora, a jej argumentami są numery kolumny i wiersza. W tym przypadku kolejne znaki będą wyprowadzane kolumnami po 16 znaków w kolumnie. Wyrażenie *j*%16 daje w wyniku resztę z dzielenia *j* przez 16. Znaki będą umieszczane w wierszach o numerach od 5 do 20 oraz w kolumnach o numerach 1, 11, 21, ..., 71.

Funkcja *cprintf* będzie tu dwukrotnie wyprowadzać kolejne wartości *z* w postaci dziesiętnej (%*d*) a po spacji w postaci znaku (%*c*).

```
getch( );
```

Funkcja *getch* pobiera z bufora klawiatury znak i w wyniku zwraca jego kod. Jeśli bufor jest pusty, funkcja czeka na naciśnięcie klawisza. W tym programie *getch*( ); zatrzymuje ekran wyjściowy programu, który po zakończeniu wykonywania się programu znika, pojawia się zaś ekran edytora. Program kończy nawias klamrowy }, zamykający funkcję *main*.

Przykład 1.2. Obliczanie pierwiastków równania kwadratowego

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main(void)
  { double a=0,b,c,d,e;
    clrscr( );
    fprintf(stderr,"Program obliczania pierwiastkow "
                   "rownania ax2+bx+c=0\n\n");
    do  {  fprintf(stderr,"A=");
           fflush(stdin);
           scanf("%lf", &a);
        } while(a==0.0);
    fprintf(stderr,"B=");
    scanf("%lf", &b);
    fprintf(stderr,"C=");
    scanf("%lf", &c);
    e=-b/(a+a);
    d=e*e-c/a;
    if(d<0.0) printf("\nBrak pierwiastkow rzeczywistych\n");
      else
        if(d==0.0) printf("\nx1,2=%lf\n", e);
         else {d=sqrt(d);
               printf("\nx1=%lf      x2=%lf\n", e-d, e+d);
              }
    getch();
  }
```

Zamiast funkcji *cprintf* wyprowadzającej teksty bezpośrednio na ekran, użyto tu funkcji *printf*, która wyprowadza je do standardowego strumienia wyjściowego *stdout*. Strumień ten jest na ogół skojarzony z ekranem, funkcja *printf* zatem wyprowadza zazwyczaj teksty na ekran. Odpowiednim wywołaniem dosowym można ten strumień

skierować do dowolnego innego urządzenia, takiego jak np. drukarka czy dysk. Dlatego zaproszenia do wprowadzania danych należy wyprowadzać funkcją *fprintf* do strumienia *stderr* a nie do *stdout*. Prototypy funkcji *printf* oraz funkcji *scanf* użytej do wprowadzania danych zawiera plik *stdio.h*. W programie użyto też funkcji pierwiastkowania *sqrt*, której prototyp znajduje się w pliku *math.h*. Oba pliki zostały włączone do programu instrukcjami

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

Definicja

```
double a=0,b,c,d,e;
```

definiuje pięć zmiennych rzeczywistych podwójnej precyzji, przy czym zmienna *a* jest inicjowana wartością zerową. Zmienne typu *float* są mało dokładne. Funkcja *sqrt*, podobnie jak pozostałe funkcje matematyczne, ma argumenty i wynik typu *double*. Jeśli nie zachodzi konieczność oszczędności pamięci, należy raczej stosować zmienne typu *double* a nie typu *float*.

W instrukcji

```
fprintf(stderr,"Program obliczania pierwiastkow "
                    "rownania ax²+bx+c=0\n\n");
```

użyto znaku drugiej potęgi o kodzie 253 oraz symbolicznych znaków \n, które nakazują przejście do nowej linii. Funkcja *printf* interpretuje znak \n jako dwa znaki \n\r, nakazujące przejście na początek nowej linii. W funkcji *cprintf* należałoby użyć obu tych znaków. Znak kwadratu można by też podać w postaci symbolicznej ósemkowej \375 (253 = 0375 w zapisie ósemkowym) lub szesnastkowej (heksagonalnej) \xFD. W systemie Windows kodowi temu odpowiada znak *ř*. Dlatego w programach przenośnych zaleca się używać tylko znaków ASCII (o kodach mniejszych od 128).

Wczytywanie wartości zmiennej *a* zorganizowano w pętli

```
do { fprintf(stderr,"A=");
     fflush(stdin);
     scanf("%lf", &a);
   } while(a==0.0);
```

która zabezpiecza tę zmienną przed wprowadzeniem do niej wartości zerowej. Funkcja *fflush* czyści bufor podanego strumienia, w tym przypadku wejściowego strumienia *stdin*. Aby dokonać konwersji wprowadzonej wartości do typu *double*, w funkcji *scanf* użyto formatu konwersji %*lf*. Parametr odpowiadający temu polu konwersji musi wskazywać zmienną *a*. Nie może to być więc zmienna *a*, lecz wyrażenie **&a**. Zawsze, gdy dany parametr ma wyprowadzać (dawać w wyniku) wartość do jakiejś zmiennej, należy posłużyć się wskaźnikiem do tej zmiennej, poprzedzając ją operatorem **&**. Wewnątrz wywoływanej funkcji należy podstawić wyprowadzaną wartość pod wskazaną zmienną.

Znak obliczonego instrukcjami $e=-b/(a+a)$; oraz $d=e*e-c/a$; wyróżnika d decyduje o istnieniu i liczbie pierwiastków rzeczywistych.

```
if(d<0.0) printf("\nBrak pierwiastkow rzeczywistych\n");
  else
    if(d==0.0) printf("\nx1,2=%lf\n", e);
     else  {d=sqrt(d);
            printf("\nx1=%lf      x2=%lf\n", e-d, e+d);
           }
```

Zależnie od wyniku porównania wyprowadza się: informację o braku pierwiastków rzeczywistych, wartość pierwiastka podwójnego lub oblicza się i wyprowadza dwa różne pierwiastki.

Przykład 1.3. Użycie funkcji i tablic

Program oblicza wartość średnią i odchylenie średniokwadratowe, czyli dyspersję.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define Max 100

void czytaj(int n, double x[], char *nazwa)
  {
    int i;
    for(i=0; i<n; i++)
      { fprintf(stderr,"%s[%d]=", nazwa, i+1);
        scanf("%lf", &x[i]);
      }
  }

double srednia(int n, double x[], double *dys)
  { int i;
    double y, z;
    *dys=0.0;
    if(n<2) return(x[0]);
    for(y=0.0, i=0; i<n; i++) y+=x[i];
    y/=n;
    for(i=0; i<n; i++) {z=x[i]-y;  *dys+=z*z;}
    *dys/=n-1;
    *dys=sqrt(*dys);
    return(y);
  }
```

```
void main(void)
  { int n;
    double s, d, x[Max];
    clrscr();
    fprintf(stderr,"Obliczanie wartosci sredniej i dyspersji");
    do  {  fprintf(stderr,"\nPodaj ile bedzie liczb ? ");
           fflush(stdin);
           scanf("%d", &n);
        } while(n<2 || n>Max);
    czytaj(n, x, "X");
    s=srednia(n, x, &d);
    printf("\nSrednia=%lf    Dyspersja=%lf\n", s, d);
    getch();
  }
```

Definicja stałej *Max*

```
#define Max 100
```

ułatwia zmianę maksymalnego rozmiaru tablicy *x*. Gdy zajdzie potrzeba zmiany programu tak, aby akceptował on większe tablice, wystarczy tylko zmienić to w powyższej definicji. Lepiej jednak zdefiniować zmienną ustaloną np.

```
const int Max=100;.
```

W programie użyto funkcji o nagłówkach

```
void czytaj(int n, double x[], char *nazwa);
double srednia(int n, double x[], double *dys);
```

Instrukcja *czytaj*(*n, x, "X"*); wczyta *n* liczb rzeczywistych i umieści je w tablicy *x*. Ponieważ sama nazwa tablicy wskazuje na jej początek, to funkcja *czytaj* potrafi zapisywać elementy tej tablicy. W zaproszeniach użyje nazwy *"X"*.

Funkcja *srednia* oblicza wartość średnią i dyspersję *n* liczb rzeczywistych zawartych w tablicy *x*. Wartość średnia jest wynikiem funkcji, natomiast dyspersja jest przekazywana pod wskazanie *dys*. Definicja parametru *dys*

```
double *dys
```

oznacza, że zmienną typu *double* jest wyrażenie **dys*, natomiast samo *dys* jest wskaźnikiem do zmiennej typu *double*. Wynikiem wyrażenia **dys* jest zmienna wskazywana przez wskaźnik *dys*. Wewnątrz funkcji operacje są zatem wykonywane na zmiennej **dys*, czyli na zmiennej *d* zdefiniowanej w funkcji *main*.

Instrukcja

```
return(y);
```

poleca wyjść z funkcji z wartością *y*, to znaczy podstawić wartość *y* pod nazwę funkcji.

W funkcji głównej *main* definicja

```
double s,d, x[Max];
```

tworzy dwie zmienne *s, d* oraz jedną tablicę *x* zawierającą *Max* elementów typu *double*. Elementy tablic w języku C numeruje się od 0. Tak więc tablica *x* zawiera elementy od *x*[0] do *x*[*Max* – 1].

Ponieważ ostatni parametr formalny funkcji *srednia* jest wskaźnikiem, parametr aktualny *&d* w wywołaniu

```
s=srednia(n, x, &d);
```

też jest wskaźnikiem i wskazuje na zmienną *d*. Wartość dyspersji zostanie więc podstawiona pod tę zmienną.

1.4. Standardowe wejście i wyjście

Prototypy (opisy nagłówków) funkcji standardowego wejścia i wyjścia zawiera plik **stdio.h**. Te funkcje otwierają i zamykają strumienie oraz przesyłają dane między programem a strumieniami. Otwierane strumienie są kojarzone z plikami (plikami dyskowymi, konsolą, drukarką itp.).

Przykładowe funkcje wykonujące operacje na plikach to:

`fopen(plik, tryb)`	– otwarcie pliku o nazwie *plik* w trybie *tryb*,
`fclose(fp)`	– zamknięcie pliku skojarzonego z *fp*,
`fseek(fp,poz,baza)`	– ustawienie pozycji pliku *fp*,
`ftell(fp)`	– określenie pozycji pliku *fp*,
`fflush(fp)`	– opróżnienie bufora pliku,
`feof(fp)`	– test, czy osiągnięto koniec pliku.

Plik może być otwarty w trybie **"r"** – do odczytu, **"r+"** – do edycji, **"w"** – do zapisu, **"a"** – do dopisywania. W trybach **"w+"** i **"a+"** plik może być też czytany.

Na przykład, poniższe instrukcje otwierają zbiór *dane.txt* do odczytu i drukarkę:

```
fp=fopen("dane.txt, "r");        fp=fopen("prn", "w");
```

W każdym programie automatycznie są otwierane pliki: wejściowy – **stdin**, wyjściowy – **stdout** i diagnostyczny – **stderr**. Strumień *stderr* jest przeznaczony do komunikacji z operatorem (np. sygnalizowanie błędów) i jest zawsze skojarzony z ekranem. Strumienie *stdin* i *stdout* są domyślnie skojarzone z klawiaturą i ekranem.

Znakowe wejście i wyjście realizują funkcje:

`fgetc(fp)`	– wprowadzenie znaku z pliku *fp*,
`fgets(bufor,n,fp)`	– wprowadzenie tekstu z pliku *fp*,
`fputc(znak,fp)`	– wyprowadzenie znaku do pliku *fp*,
`fputs(bufor, fp)`	– wyprowadzenie tekstu do pliku *fp*.

Formatowane wejście i wyjście realizują funkcje:

`fscanf(fp, format, parametry)`	– wprowadzenie wartości z *fp*,
`fprintf(fp, format, parametry)`	– wyprowadzenie do *fp*.

Ponieważ strumienie *stdin* i *stdout* są bardzo często używane, opracowano dla nich odpowiedniki powyższych funkcji wejścia i wyjścia. Są to funkcje:

```
getchar();      gets(bufor);    scanf(format, parametry);
putchar(znak);  puts(bufor);    printf(format, parametry);
```

Format funkcji *fscanf* i *scanf* jest tekstem zawierającym **wyłącznie** wzorce konwersji dla kolejnych parametrów wejściowych. Każdy wzorzec konwersji ma postać **%wt**, gdzie *w* jest maksymalną liczbą znaków wprowadzanego pola, a *t* jest typem konwersji. Pole kończy się białym znakiem nie dalej jednak niż po *w* znakach. Ograniczenie długości pola można pominąć.

Konwersje typu **d**, **o**, **x** wprowadzają liczby całkowite, traktując je domyślnie jako dziesiętne, ósemkowe lub szesnastkowe. Konwersje typu **e**, **f**, wprowadzają liczby rzeczywiste (*float*). Poprzedzenie typu konwersji literą **l** oznacza, że konwersja dotyczy liczby długiej (*long* lub *double*). Konwersje typu **c** wprowadza znak, a konwersja typu **s** wprowadza tekst. Na przykład format "%6d%3d%d" zinterpretuje ciąg znaków "12 345678 9" jako liczby: 12, 345 oraz 678.

Format funkcji *fprintf* i *printf* jest wzorcem wyprowadzanego tekstu, w którym elementy zmienne (zależne od wartości wyprowadzanych argumentów) zastąpiono wzorcami konwersji w postaci **%fw.pt**, gdzie *w* jest minimalną szerokością wyjściowego pola znakowego, *p* jest precyzją, *t* jest typem konwersji, a *f* określa sposób uzupełniania pola do *w* znaków. Wszystkie elementy za wyjątkiem % oraz *t* można pominąć.

Podobnie jak w formacie wejściowym stosuje się typy konwersji **d**, **o**, **x**, **e, f**, **c**, **s**. Wyprowadzając liczby typu *long* lub *double*, należy typ konwersji poprzedzić literą **l**.

Precyzja *p* w konwersjach **e**, **f** określa liczbę cyfr po kropce, a w konwersji **s** – maksymalną liczbę wyprowadzanych znaków. Jeśli określono *w*, to wyprowadzane pole jest uzupełniane zazwyczaj spacjami lewostronnie do *w* znaków. Gdy *f* jest znakiem – (minus), uzupełniające spacje dopisywane są z prawej strony. Gdy *f* jest zerem, to przy wyprowadzaniu liczby, jej pole znakowe jest lewostronnie uzupełniane zerami.

Na przykład, jeżeli $d=5$, $m=3$, $r=1997$, a p wskazuje na tekst *Borland*, to poniższe instrukcje wyprowadzą

```
printf("|%2d-%02d-%d|", d, m, r);        | 5-03-1997|
printf("|%.3s|%-5.3s|%5.3s|",p,p,p);     |Bor|Bor   |   Bor|
```

Przykład 1.4. Współpraca z plikami

Przykładowy program wczytuje liczby całkowite dziesiętne z pliku *dane.txt* na dyskietce *A*: i wypisuje je w tabelce w postaci dziesiętnej, ósemkowej i szesnastkowej na ekranie oraz ewentualnie na drukarce.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
void main(void)
  { int n;
    FILE *fp1, *fp2;
    char *tekst1=
        "\n| Dziesietnie | Osemkowo | Szesnastkowo |"
        "\n+-------------+----------+--------------+",
           *tekst2=    "\n|%10d   |%8o  |%9x     |";
    fp1=fopen("a:dane.txt", "r");
    if(fp1==NULL)
        {  fprintf(stderr, "\nBrak pliku danych !!\n");
           goto e1;
        }
    fprintf(stderr, "\nCzy jest drukarka? (T/N) ");
    do n=toupper(getch()); while(n!='T' && n!='N');
    fp2=(n=='T')?fopen("prn", "w"):NULL;
    if(fp2) fprintf(fp2, tekst1);
    while (fscanf(fp1, "%d", &n)==1)
        {  printf(tekst2, n, n, n);
           if(fp2) fprintf(fp2, tekst2, n, n, n);
        }
    if(fp2) fclose(fp2);
    fclose(fp1);
e1:return;
  }
```

Funkcja *fscanf* zwraca liczbę prawidłowo wykonanych konwersji. W tym przykładzie funkcja ta powinna dokonać jednej konwersji. Poprawność odczytu sprawdza wyrażenie *fscanf*(*fp*1, "%*d*", &*n*)==1 .

1.5. Inne wybrane funkcje biblioteczne

Biblioteka kompilatora Borland C++ zawiera między innymi: funkcje matematyczne, funkcje obsługi ekranu, funkcje operacji na znakach i na tekstach.

1.5.1. Funkcje matematyczne

Zdecydowana większość funkcji matematycznych wymaga włączenia do programu pliku **math.h**. Te funkcje są typu *double* i mają 1 lub 2 argumenty typu *double*.

Funkcje trygonometryczne i hiperboliczne:

sin(x)	sinus,
cos(x)	cosinus,
tan(x)	tangens,
sinh(x)	sinus hiperboliczny,
cosh(x)	cosinus hiperboliczny,
tanh(x)	tangens hiperboliczny,
asin(x)	0 ### asin(x) ### ###,
acos(x)	–###2 ### acos(x) ### ###2,
atan(x)	–###2 < atan(x) < ###2,
atan2(x,y)	–### ### atan2(x, y) ### ###.

Funkcje sin(x), cos(x) i tan(x) wymagają argumentu x w radianach

Logarytmy, pierwiastki, potęgi, moduł, reszta z dzielenia

log(x)	logarytm naturalny,
log10(x)	logarytm dziesiętny,
exp(x)	eksponenta: e^x,
pow(x, y)	x^y,
ldexp(x, n)	$x \cdot 2^n$ (*int n*),
sqrt(x)	pierwiastek kwadratowy,
fabs(x)	moduł (wartość bezwzględna),
lmod(x, y)	reszta z dzielenia x / y.

Argumenty i wyniki powyższych funkcji są typu *double*.

Zaokrąglenia całkowite

double ceil(double x)	*ceil*(x) ### x, np. *ceil*(4.1) = 5.0,
double floor(double x)	*floor*(x) ### x np. *floor*(4.9) = 4.0.

Funkcje obliczania wartości bezwzględnej liczb typu *int* i *long* oraz **generator liczb losowych** wymagają dołączenia pliku **stdlib.h**

`int abs(int k)`	moduł // *abs*(–32768) = –32768,
`long labs(long k)`	moduł,
`int rand(void)`	liczba losowa od 0 do 32767.

Przykłady

Liczbę ### oraz liczę e można wyznaczyć instrukcjami

```
Pi=4.0*atan(1.0);          E=exp(1.0);
```

W końcowej części pliku *math.h* znajdują się definicje wielu stałych matematycznych takich jak π (*M_PI*), e (*M_E*) i inne.

Tabela 1.1. Stałe zdefiniowane w pliku *math.h*

Wartość	Nazwa stałej	Wartość definiowana
e	M_E	2.71828182845904523536
$\log_2 e$	M_LOG2E	1.44269504088896340736
$\log e$	M_LOG10E	0.43429448190325182765 1
$\ln 2$	M_LN2	0.69314718055994530941 7
$\ln 10$	M_LN10	2.30258509299404568402
###	M_PI	3.14159265358979323846
###2	M_PI_2	1.57079632679489661923
###4	M_PI_4	0.78539816339744830961 6
1/###	M_1_PI	0.31830988618379067153 8
2/###	M_2_PI	0.63661977236758134307 6
$1/\sqrt{\pi}$	M_1_SQRTPI	0.56418958354775628694 8
$2/\sqrt{\pi}$	M_2_SQRTPI	1.12837916709551257390
$\sqrt{2}$	M_SQRT2	1.41421356237309504880
$\sqrt{0{,}5}$	M_SQRT_2	0.70710678118654752440 1

Liczbę rzeczywistą *x* (typu *double* lub *float*) można zaokrąglić do *n* cyfr po kropce dziesiętnej lub *n* cyfr przed kropką za pomocą instrukcji:

`a=pow(10.0, (double)n);`	$a = 10^n$,
`x=floor(a*x+0.5)/a;`	zaokrąglenie po kropce,
`x=a*floor(x/a+0.5);`	zaokrąglenie przed kropką.

Liczby losowe z przedziału [0, 1) lub [0, *a*) można generować za pomocą instrukcji:

`r1=rand()*ldexp(1.0, -15);`	$0 \le r1 < 1$,
`r2=rand()*ldexp(a, -15);`	$0 \le r2 < a$.

1.5.2. Funkcje obsługi ekranu

Prototypy poniższych funkcji ekranowych zawiera plik **conio.h**

`clrscr()`	– czyszczenie ekranu (okna),
`clreol()`	– czyszczenie od kursora do końca linii w oknie,
`gotoxy(x,y)`	– ustawienie kursora w kolumnie *x* i wierszu *y*,
`wherex()`	– odczytanie współrzędnej *x* kursora,
`wherey()`	– odczytanie współrzędnej *y* kursora,
`kbhit()`	– test obecności znaku w buforze klawiatury,
`getch()`	– test i pobranie znaku z bufora klawiatury,
`textbackground(color)`	– ustawienie koloru tła,
`textcolor(color)`	– ustawienie koloru tekstu,
`window(x1,y1,x2,y2)`	– ustawienie okna,
`gettext(x1,y1,x2,y2,buf)`	– zapamiętanie fragmentu ekranu,
`puttext(x1,y1,x2,y2,buf)`	– wyprowadzenie fragmentu na ekran,
`cprintf(format, param)`	– formatowane wyjście na ekran.

Przykład 1.5

Program ilustruje sposób wczytywania kodu klawisza, pozycjonowania tekstu na ekranie i zapętlenia programu.

```
#include <conio.h>
#include <ctype.h>
void main(void)
  { int n, z;
    clrscr();                          // czyszczenie ekranu
    n=1;                               // licznik wykonań
    do                                 // początek pętli 1
    { gotoxy(20,10);                   // pozycja: znak 20, linia 10
      cprintf("Wykonanie nr. %d", n++);
      gotoxy(1,25);                    // lewy dolny róg ekranu
      cprintf("Czy koniec (T/N)");
      while(kbhit()) getch();          // opróżnienie bufora
      do                               // początek pętli 2
        z=toupper(getch());            // czytanie klawisza
        while(z!='T' && z!='N');       // koniec pętli, gdy z=T/N
      gotoxy(1,25);                    // lewy dolny róg ekranu
      clreol();                        // usuń tekst z linii
    } while(z=='N');                   // koniec pętli 1, gdy z=T
  }
```

Klawisze funkcyjne wprowadzają do bufora klawiatury zero przed kodem klawisza. Tak więc, gdy funkcja *getch*( ) zwróci zero, należy wywołać ją powtórnie, aby odczytać kod klawisza funkcyjnego.

Przykład 1.6. Operowanie kolorami tekstu – menu

Program wypisuje na ekranie 6 pozycji w układzie menu i podświetla pierwszą pozycję. Wyboru pozycji można dokonać używając klawiszy 1, 2, 3, 4, 5 lub 6. Klawisze *Enter* oraz *Esc* kończą program. W programie użyto funkcji **cprintf**, która wyprowadza dane bezpośrednio na ekran. Funkcja *printf* wyprowadziłaby tekst na ekran za pośrednictwem strumienia *stdout* bez atrybutów, to znaczy bez zmiany kolorów.

```
#include <conio.h>

void main(void)
  {
    int n, M, z, z1;
    char *t[]={"1-jeden","2-dwa","3-trzy","4-cztery",
                      "5-piec","6-szesc"};
    M=sizeof(t)/sizeof(t[0]);        // wyznaczenie rozmiaru tablicy t
    textbackground(BLACK);
    clrscr();
    window(30, 8, 50, 9+M);
    textbackground(YELLOW);
    clrscr();
    window(32, 9, 48, 8+M);
    textbackground(BLUE);
    textcolor(YELLOW);
    clrscr();
    for(n=0; n<M; n++)               // pokazanie menu
        {
           gotoxy(2, n+1);
           cprintf(t[n]);
        }
    z=z1=1;
    while(kbhit()) getch();          // opróżnienie bufora klawiatury
    do {
        gotoxy(2, z1);
        cprintf(t[z1-1]);            // tekst w normalnych kolorach
        textbackground(GREEN);
        textcolor(BLACK);
        gotoxy(2, z);
        cprintf(t[z-1]);             // tekst w odwróconych kolorach
```

```
      textbackground(BLUE);
      textcolor(YELLOW);
      z1=z;
      do z=getch();                  // kod naciśniętego klawisza
         while(z!=27 && z!='\r' && z<'1' || z>(M+'0'));
      z-='0';                        // naciśnięta cyfra
    } while (z>0 && z<=M);           // wybieraj aż Esc lub Enter
  window(1, 1, 80, 25);
  gotoxy(1, 22);
  if(z=='\r'-'0') cprintf("Wybrano: %s", t[z1-1]);
        else  cprintf("Wycofano akcje - Esc");
  getch();
}
```

1.5.3. Funkcje operacji na znakach i na tekstach

Funkcje z prototypami w pliku **ctype.h**:

toupper(c) – konwersja małej litery na dużą,
tolower(c) – konwersja dużej litery na małą
isalpha(c) – badanie, czy *c* jest literą,
isdigit(c) – badanie, czy *c* jest cyfrą,
isprint(c) – badanie, czy *c* jest drukowalne,
isspace(c) – czy *c* jest jest widoczne.

Funkcje z prototypami w pliku **string.h**

strcpy(dest,source) – kopiowanie tekstu z *source* do *dest*,
strcat(dest,source) – do tekstu w *dest* dołączenie tekstu z *source*,
strlen(str) – obliczenie długości tekstu *str*,
strcmp(str1,str2) – porównanie dwu tekstów,
strchr(str1,znak) – odszukanie znaku *znak* w tekście *str*1.

Przykład 1.7: Zmiana wielkości liter na duże.

Program zmienia wprowadzoną nazwę pliku na nazwę z rozszerzeniem BAK. Jeśli podana nazwa nie zawiera rozszerzenia, to zostanie ono dopisane. Wszystkie litery nazwy zostaną zamienione na duże.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>

void main(void)
  {
    char buf[80], *p, B[]=".BAK";
    printf("\n\n\n Podaj nazwe pliku: ");
    scanf("%79s", buf);
    p=strchr(buf, '.');
    if(p==NULL) strcat(buf, B);
                        else strcpy(p, B);
    for(p=buf; *p; p++)
                         *p=toupper(*p);
    printf("%s", buf);
    getch();
  }
```

Aktualne dla danego kompilatora informacje o tych i pozostałych funkcjach otrzymuje się po naciśnięciu klawiszy F1 lub Ctrl-F1.

Pytania i zadania

1.1. Podaj cechy wspólne języka C z językiem PASCAL.

1.2. Na czym polega blokowa struktura języka C i języka Pascal?

1.3. Co to są zmienne globalne, zmienne lokalne i jaki jest ich zasięg?

1.4. Wymień uproszczenia logiki języka C w porównaniu z logiką języka Pascal.

1.5. Jakie są w konsekwencje traktowania w języku C wartości znakowych i wartości logicznych jako wartości liczbowych?

1.6. Czym w sensie języka C są tablice wieloindeksowe?

1.7. Napisz instrukcje, które obliczą:

a) $c = \sqrt{a^2 + b^2}$ b) $c = \sqrt{a^2 + b^2 - 2ab\cos\gamma}$

c) $r = \sqrt[3]{\frac{3V}{4\pi}}$ d) $K = A\left(1 + \frac{p}{100}\right)^n$

Zadania laboratoryjne

1.8. Przygotuj własną kartotekę, tak jak opisano na początku rozdziału, a następnie wpisz i uruchom program z przykładu 1.1.

1.9. Wpisz program z przykładu 1.2 i uruchom go. Przenieś rozkazy drukowania w instrukcjach *if* do nowych linii, tak aby warunek *if* i funkcja *printf* były w odrębnych liniach. Prześledź wykonanie programu używając klawisza F7 lub F8.

1.10. W programie z przykładu 1.2 ustaw kursor na pierwszym warunku *if* i uruchom program klawiszem F4. Gdy program zatrzyma się, naciśnij Ctrl F4 i odczytaj wartość zmiennej *d*. Wpisz nową wartość i kontynuuj śledzenie.

1.11. Uruchom program z przykładu 1.3. Prześledź wykonanie tego programu za pomocą klawisza F8, a następnie za pomocą klawisza F7. Jaka jest różnica między śledzeniem pierwszym a drugim?

1.12. Dowolnym sposobem (F4, F7 lub F8) zatrzymaj program z przykładu 1.3 w dowolnym miejscu. Najedź kursorem na dowolną zmienną i odczytaj jej wartość klawiszem Alt F4. Tym sposobem odczytaj zawartość tablicy *x*.

1.13. Wykonując program z przykładu 1.3, prześledź wartości zmiennych *s, d, y* używając okna obserwacyjnego (Alt-WW).

1.14. Za pomocą klawiszy Ctrl F1 zapoznaj się z opisem kilku wybranych funkcji.

1.15. Uruchom program 1.4, który czyta dane z dyskietki. Przerób ten program tak, aby: a) nazwa pliku z danymi mogła być wprowadzana z konsoli, b) wyniki były wyprowadzane do pliku dyskowego.

1.16. Program 1.6 uzupełnij tak, aby: a) reagował na klawisze strzałek, b) przed zakończeniem odtwarzał poprzednią zawartość tej części ekranu, która została zmieniona przez menu.

1.17. Napisz program, który po naciśnięciu *Enter* umieści na ekranie pusty prostokąt o 16 kolumnach i 5 wierszach. Po naciśnięciu *Esc* należy przywrócić poprzednią zawartość ekranu.

2. Stałe i zmienne

Programy w języku C zawierają następujące jednostki leksykalne: identyfikatory, słowa kluczowe, stałe i teksty, operatory oraz separatory (w tym komentarze). **Znaki białe (niewidoczne) są ignorowane, chyba że rozdzielają jednostki leksykalne. Jednostką leksykalną jest najdłuższy ciąg znaków, z których można utworzyć jednostkę**. Na przykład *A+++B* jest rozumiane jako (*A*++) + *B*, natomiast *A+ ++B* jest rozumiane jako *A* + (++*B*). Wyrażenie *a/ *b* oznacza dzielenie *a* przez **b*, natomiast w tekście *a/*b* znaki /* rozpoczną komentarz.

Komentarz rozpoczyna się znakami /*, a kończy znakami */ lub zaczyna się znakami //, a kończy znakiem nowej linii. Komentarz jest traktowany jak pojedyncza spacja.

Stałymi są liczby całkowite lub rzeczywiste. Stałe całkowite mogą być przedstawione w postaci znakowej, dziesiętnej, ósemkowej lub heksagonalnej.

Zmienne są obiektami, w których przechowuje się wartości różnych typów (np.: wartości liczbowe rzeczywiste, całkowite lub wskaźniki). Zmienne identyfikuje się przez ich identyfikatory (nazwy).

2.1. Identyfikatory zmiennych

Identyfikator jest ciągiem liter i cyfr. Pierwszym znakiem musi być litera, przy czym znak podkreślenia _ uważa się za literę. **Słowa kluczowe języka C**, takie jak:

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

oraz dodatkowe słowa kluczowe kompilatora Borland:

asm	**catch**	**cdecl**	**class**	**delete**	**far**	**fortran**
friend	**huge**	**inline**	**interrupt**	**new**	**near**	**operator**
pascal	**private**	**protected**	**public**	**template**	**this**	**virtual**
_cs	**_ds**	**_es**	**_ss**			

nie mogą być identyfikatorami. Kompilator Borland predefiniuje też nazwy rejestrów:

_AX	**_AL**	**_AH**	**_BX**	**_BL**	**_BH**	**_CX**	**_CL**	**_CH**	**_DX**
_DL	**_DH**	**_CS**	**_DS**	**_SS**	**_ES**	**_SP**	**_BP**	**_DI**	**_SI**

W języku C rozróżnia się małe i duże litery. Tak więc słowa *Char* lub *Int* mogą być identyfikatorami, natomiast słowa *char* oraz *int* nie mogą być, gdyż są one słowami kluczowymi.

Dozwolona długość identyfikatora zależy od implementacji. Na przykład Borland C++ 2.0 dopuszcza 32 znaki. Standard języka gwarantuje, że identyfikatory różniące się na pierwszych ośmiu znakach oznaczają różne obiekty. Tak więc np. w implementacji TURBO C identyfikatory *Identyfikator_A* i *Identyfikator_B* są utożsamiane z jednym obiektem. Dobrym obyczajem jest używanie takich identyfikatorów, które opisują oznaczane obiekty.

Poprawne identyfikatory: A, Pi2, Nr_wiersza, USD, Auto, H20a, _max, _2, zl.

Błędne identyfikatory: @A, 2Pi, Nr-wiersza, US$, auto, 3_max, zł.

Pytania i zadania

2.1. Które z tych ciągów są identyfikatorami?

a) `Program` b) `zakończenie` c) `gamma3` d) `###3`
e) `beta4` f) `beta 4` g) `beta-4` h) `beta_4`
i) `nr.kol.` j) `_2wiersz` k) `2_wiersz` l) `A_kwadrat`

2.2. Typy i rozmiary danych

Danymi w języku C są liczby całkowite i liczby rzeczywiste. Znaki są liczbami całkowitymi z przedziału od 0 do 255. Wyniki operacji logicznych i porównań są liczbami całkowitymi 0 lub 1. Dane mogą być przedstawiane w pamięci w różny sposób. Sposób ten określony jest przez **typ** danych. Dane całkowite mogą w każdym typie przedstawiać liczbę ze znakiem (*signed*) lub liczbę nieujemną bez znaku (*unsigned*). Zestawienie typów danych wraz z ich przedziałami wartości

i rozmiarami, czyli liczbami bajtów zajmowanych przez dane tych typów dla kompilatora Borland C++ przedstawiono w tabeli 2.1.

Rozmiary (*sizeof*) i zakresy danych poszczególnych typów mogą być różne dla innych implementacji. Wymaga siê jednak, aby

sizeof(char) ### sizeof(short) ### sizeof(int) ### sizeof(long),
sizeof(float) ### sizeof(double) ### sizeof(long double).

Tabela 2.1. Typy podstawowe

Dane	Typ danych	Przedział wartości		Rozmiar w bajtach
		signed	**unsigned**	
Całkowite	**char**	–128 ### 127	0 ### 255	1
	short	–32768 ### 32767	0 ### 65535	2
	int	–32768 ### 32767	0 ### 65535	2
	long	-2^{31} ### $2^{31}-1$	0 ### $2^{32}-1$	4
Rzeczywiste	**float**	$-3.4\cdot10^{38}$ ### $3.4\cdot10^{38}$		4
	double	$-1.7\cdot10^{308}$ ### $1.7\cdot10^{308}$		8
	long double	$-1.7\cdot10^{4932}$ ### $1.7\cdot10^{4932}$		10

Przekraczanie zakresu zmiennych całkowitych jest częstą i trudną do wykrycia przyczyną nieprawidłowego działania programu.

Należy zwrócić uwagę, że czterobajtowe liczby rzeczywiste typu *float* są mało dokładne i w obliczeniach powinno się raczej używać liczb typu *double*.

Pytania i zadania

2.2. Skróć nazwy typów
a) signed int b) unsigned int c) unsigned long int
d) signed long e) short int f) signed short

2.3. Jakie typy mogą mieć zmienne, które oznaczają:
a) odległości pomiędzy miastami w km,
b) odległości planet od słońca w km,
c) masy właściwe w kg/m^3,
d) liczbę uczniów w klasie,
e) średnie oceny z przedmiotów.

2.4. Jaki jest związek między przedziałem wartości typu całkowitego a rozmiarem typu w bajtach?

2.3. Stałe i teksty

Stałe dzielą się na: znakowe, całkowite, rzeczywiste, łańcuchowe (teksty).

Tabela 2.2. Stałe w języku C

Stała	Postać	Przykłady
Znakowa	literał	'A', 'a', ' ', '1', '2',
	symboliczny kod	'\101', '\0x41', '\375', '\15'
	symboliczny opis	'\n', '\t', '\b', '\r', '\f', '\a', '\\', '\'', '\"', '\?'
Całkowita	dziesiętna	46, 46L, 46l, 46U, 46u, 46ul, 46UL
	ósemkowa	056, 056L, 056l, 056u, 056U, 056ul, 056UL
	heksagonalna	0x2E, 0X2E, 0x2eL, 0x2eu, 0x2Eul, ox2EUL
Rzeczywista	dziesiętna	46.0, 0.078, 0.0, 39.2F, 89.4L, 6520L
	wykładnicza	4.6e1, 7.8E–2, 3.92e1F, 894e–1L, 65.2E2L
Łańcuchowa		"Jezyk C", "\nRownanie ax\375 + bx + c = 0\n"

Literał jest pojedynczym znakiem ujętym w apostrofy. Wartość literału jest równa kodowi (zwykle ASCII) znaku. Typ wartości zależy od implementacji i jest *int* lub *char*.

Symboliczny kod, to ujęta w apostrofy i poprzedzona znakiem \ (back slash) liczba ósemkowa lub heksagonalna. Liczba ta podaje kod znaku i jest typu *unsigned char*.

Symboliczny opis ma w miejsce kodu literowe oznaczenie znaku. Podane w tabeli symbole oznaczają: **\n** – nowa linia, **\t** – tabulacja, **\b** – backspace, **\r** – początek wiersza, **\f** – nowa strona, **\a** – sygnał dzwonka oraz \\ **\'** **\"** **\?** – odpowiednio znaki \ ' " ?.

Stałe znakowe są liczbami całkowitymi. Można ich zatem użyć wszędzie tam, gdzie można używać liczb całkowitych. Na przykład:

'A'+3 jest równe 68 i daje znak *D*,
'M'+'a'–'A' zamienia duże *M* na małe *m*,
'M'+32 zamienia duże *M* na małe *m*,
'a'–'A' jest równe 32 czyli znakowi spacji.

Stała całkowita może być zakończona małymi lub dużymi literami U, L, UL i wtedy jest ona typu odpowiednio *unsigned, long, unsigned long*. W przeciwnym razie typ stałej zależy od jej wartości, jak pokazano w tabeli 2.3.

Tabela 2.3. Typy stałych całkowitych

Najwęższy typ, w którym mieści się wartość stałej	Typ stałej	
	ósemkowej, heksagonalnej	dziesiętnej
`int`	`int`	
`unsigned`	`unsigned`	`long`
`long`	`long`	
`unsigned long`	`unsigned long`	

Stała rzeczywista bez przyrostka jest typu *double*. Stałe rzeczywiste zakończone małymi lub dużymi literami F, L są odpowiednio typu *float, long double*.

Stałe zawsze są bez znaku. Tak więc napis –32768 nie jest liczbą ujemną typu *int*, ale wyrażeniem złożonym z operacji minus i liczby dodatniej typu *long*. Podobnie jak stałe całkowite, stałe rzeczywiste są nieujemne. Poprzedzone znakiem – (minus) tworzą wyrażenie.

Stałą łańcuchową (tekstem) jest ciąg znaków zamknięty w cudzysłowy. Stała ta może mieć dowolną długość i zawiera na końcu znak \0, który jest ogranicznikiem tekstu. Na przykład tekst „Jezyk C" ma siedem znaków (długość tekstu = 7), ale zajmuje on osiem bajtów.

J	e	z	y	k		C	\0

Teksty mogą być umieszczane w tablicach typu *char*. **Rozmiar tablicy musi uwzględniać miejsce na ogranicznik '\0'.** Stała łańcuchowa zawierająca *N* znaków jest typu *char*[*N*+1].

Każda para sąsiadujących stałych łańcuchowych jest traktowana jako jeden łańcuch. Na przykład napisy

```
"Jezyk C",
"Jezyk"" C",
"Jez""yk\40C"
```

są równoważne, ale napisy

```
"A"    i    'A'
```

są różne. Napis *"A"* jest dwubajtowym tekstem typu *char*[2] złożonym ze znaku '*A'* i znaku '\0', natomiast napis *'A'* jest literałem o wartości 65. Wartością łańcucha "*A*" jest wskazanie typu *char** na początkowy znak tego łańcucha. Wartością literału *'A'* jest liczba całkowita 65 typu *int* w języku C lub typu *char* w języku C++.

Pytania i zadania

2.5. Jak będzie wyglądać na ekranie wyprowadzony funkcją *cprintf*, a jak funkcją *printf* tekst.

```
a) "\n\rPiszemy\n\" programy wC\"\15\r Jezyku C."
b) "\n\rTen tekst\n\"\\a\"\ngeneruje pisk."
```

2.6. Jakie są długości i typy tekstów z zadania 2.5?

2.7. Jakie są wartości i typy stałych (w implementacji Borland C++)?

a) 20	b) 20.0	c) 2e1	d) 2UL	e) 2.L
f) 020	g) 0x20	h) 3.5e3	i) 7.8E–6	j) 2.E7F
k) 3.14	l) 0x2E	m) 0xful	n) 0xF	o) 0.F

2.8. Jakie są wartości i typy stałych w implementacji Borland C++?

a) 41520	b) 0141524	c) 0x4150	d) 0x8012
e) 31510	f) 0200212	g) 0xA00A0022	h) 0x700A0022

2.9. Które z napisów są stałymi w języku C?

a) 0EL	b) 0XEL	c) –5.0	d) 5e–2	e) –5e2
f) "Beta"	g) "A"	h) 'A'	i) 'Beta'	j) 0uL
k) 0xU	l) 0.75L	m) 0.75u	n) 0.75ul	o) 0x7.5

2.10. Jakie są typy stałych łańcuchowych?

a) `"Jezyk C"`	b) `"\nJezyk C\n"`	c) `"Jezy""k C"`
d) `"Jezyk\40\103"`	e) `"Jez","yk C"`	f) `"\"Borland C++\""`
g) `""`	h) `"Je" "zy" "k C"`	i) `"\n\rBorland C++"`

2.4. Definicje i deklaracje zmiennych

Definicje i deklaracje w języku C służą do interpretacji identyfikatorów i mają postać

<Specyfikator klasy> <Typ> <Lista deklaratorów> ;

W przeciwieństwie do definicji deklaracja nie tworzy zmiennej, nie rezerwuje dla niej pamięci, lecz tylko informuje, czym jest deklarowany identyfikator.

Jeśli specyfikator klasy nie występuje, to dla definicji globalnych przyjmuje się klasę *extern*, a dla definicji lokalnych klasę *auto*.

Przykładowe definicje zmiennych prostych typu *double* bez specyfikacji klasy:

```
double x, y, z;
```

Jeśli w liście deklaratorów identyfikator jest poprzedzony operatorem wyłuskania * (gwiazdka), to deklarowana zmienna jest typu wskazującego. Na przykład:

```
double *px;
```

definiuje zmienną *px*, która może wskazywać (czyli zawierać programowy adres) na zmienne typu *double*. Zmienna *px* jest wskaźnikiem typu *double**.

2.4.1. Zasięg definicji i deklaracji

Zasięgiem definicji lub deklaracji identyfikatora jest część programu od miejsca zdefiniowania lub zadeklarowania do końca najwęższego bloku obejmującego tę definicję lub deklarację. Dla identyfikatorów globalnych lub zewnętrznych zasięg rozciąga się do końca pliku.
Jeśli w obszarze zasięgu identyfikatora wystąpi w bloku wewnętrznym definicja lub deklaracja identyfikatora o tej samej nazwie, to przesłania ona w tym bloku identyfikator zewnętrzny.
Zasięg zmiennych globalnych obejmuje plik, w którym te zmienne są zdefiniowane.

Na przykład

```
    {
1.   int K=1,N;
      . . .
2.    {int K=8
       N:N=0;
     . . .
3.    }
      . . .
4.  }
```

Zasięg zmiennej $K = 1$

Zasięg zmiennej $K = 8$

Zasięg definicji *int* $K=1$; rozciąga się od punktu 1 do 2 oraz od punktu 3 do 4. W bloku wewnętrznym definicja $K=1$ jest przesłonięta definicją *int* $K=8$, której zasięg rozciąga się od punktu 2 do 3. Zmienna *N* obejmuje swoim zasięgiem cały obszar od punktu 1 do 4, pomimo że w bloku wewnętrznym występuje etykieta *N* w instrukcji *N:N*=0;, ponieważ identyfikatory zmiennych i etykiet nie przykrywają się.

Pytania i zadania

2.11. Zaznacz zasięgi definicji w podanych programach. Co wydrukują te programy?

a)
```
#include <stdio.h>
int R=5;
void main (void)
{int R=30;
  {int R=100;
   printf("R3=%d ", R);
  }
 printf(R2=%d", R);
}
```

b)
```
#include <stdio.h>
int K=3;
void funkcja (void)
 {printf("K1=%d\n", K);}
void main (void)
 {int K=20;
  printf("K2=%d", K);
  funkcja( );
 }
```

c)
```
#include <stdio.h>
int K=3;
int funkcja (void);
  { return (K);}
void main (void)
{int K=20;
 printf("K=%d", K+fun());
}
```

d)
```
#include <stdio.h>
int R=5;
void main (void)
{int R=30;
 {int R=100;
  printf("R3=%d", R+::R);
 }
 printf(R2=%d", ::R+R);
}
```

2.4.2. Klasy pamięci

Istnieją cztery klasy pamięci: **auto** (automatyczna), **static** (statyczna), **extern** (zewnętrzna) i **register** (rejestrowa) oraz **typedef** (definicja typu).

Zmienne automatyczne (auto) są lokalne w swoim bloku i znikają, gdy sterowanie opuści ten blok. Zmienne te powstają i są inicjowane wartościami początkowymi przy każdym uaktywnianiu bloku.

Zmienne statyczne (static) są lokalne w swoim bloku, ale istnieją przez cały czas wykonywania się programu i zachowują swoje wartości do ponownego uaktywnienia bloku, nawet gdy sterowanie już ten blok opuściło. Inicjowane wartości początkowe nadaje się tym zmiennym w momencie ich powstania, raz przed rozpoczęciem wykonywania programu.

Zmienne zewnętrzne (extern) istnieją i zachowują swoje wartości podczas wykonywania całego programu, można ich więc używać do komunikacji między funkcjami, nawet oddzielnie kompilowanymi. Podobnie jak zmienne statyczne powstają i są inicjowane wartościami początkowymi raz przed rozpoczęciem wy-

konywania programu. Deklaracja ze specyfikacją *extern* nie rezerwuje pamięci (nie jest definicją), lecz informuje, że wymienione identyfikatory są definiowane gdzieś na zewnątrz, np. w innym pliku.

Zmienne rejestrowe (register) umieszcza się (o ile to możliwe) w szybkich rejestrach maszyny. Podobnie jak zmienne automatyczne są one lokalne dla każdego bloku i znikają po zakończeniu wykonywania bloku. Mogą być nimi tylko zmienne typu *char*, *int* lub wskaźnik. Zmienne rejestrowe nie mają adresów.

Deklaracja typu (typedef) nie rezerwuje pamięci. Informuje, że deklarowane identyfikatory posłużą jako identyfikatory typów.

Deklaracja zmiennej zewnętrznej (extern) nie przydziela tej zmiennej pamięci (nie definiuje tej zmiennej). Deklaracja ta wskazuje, że deklarowany identyfikator jest definiowany w innym pliku. Zasięg definicji poszerza się więc o zasięg deklaracji ze specyfikacją extern.

Przykłady definicji i deklaracji:

```
extern int MAX,MIN;
static double XX;
register int j;
double x,y,delta,x_kwadrat;
unsigned long k1,k2,k3;
float fs;
```

2.5. Tablice

Tablice definiuje się podając liczbę elementów. Liczba ta musi być stałą całkowitą. Definicje

```
char Naz1[80], tx2[32];
float TF[120];
```

definiują tablice 80- i 32-elementowe typu *char* oraz tablicę 120 liczb typu *float*. Tablice *Naz*1, *tx*2, *TF* są kolejno typu: *char*[80], *char*[32] i *float*[120].

W języku C elementy *N*-elementowej tablicy numeruje się od 0 do *N* – 1.

Tablice wielowymiarowe są faktycznie jednowymiarowymi tablicami, których elementami są też tablice, np.

```
double S[10][15];
```

definiuje dziesięcioelementową tablicę, której każdy element jest tablicą 15 elementów typu *double*. Elementami tablicy *S* są tablice *S*[0], *S*[1], ..., *S*[9]. Elementami tablicy *S*[*i*] (*i*=0,1, ..., 9) są zmienne typu *double* *S*[*i*][0], *S*[*i*][1], ..., *S*[*i*][14]. *S*[*i*][*j*] jest typu *double*. *S*[*i*] jest typu *double*[15], a *S* jest typu *double*[10][15].

Para nawiasów [ ] jest dwuargumentowym lewostronnie łącznym operatorem indeksacji, którego wynikiem jest element tablicy. Argumentami są: wskaźnik do tablicy (np. *S*) i numer elementu.

Napis *S*[*i*][*j*] jest wyrażeniem wykonywanym jako (*S*[*i*])[*j*]. Najpierw wyznaczany jest *i*-ty element tablicy *S*. Ten element (*S*[*i*]) jest traktowany jak wskaźnik do (nazwa) 15-elementowej tablicy. Kolejna operacja [ ] wyznacza element tej tablicy.

Operator indeksacji [ ] ma wyższy priorytet niż operator wyłuskania *. Tak więc definicja

```
float *R[20];
```

definiuje tablicę 20 wskaźników, bowiem *R* najpierw się wiąże z operatorem [ ].

Aby zdefiniować wskaźnik do tablicy, należy powiązać definiowany identyfikator najpierw z operatorem wyłuskania *, a potem z operatorem indeksacji [ ], np.

```
float (*P)[8];
```

W tym przypadku **w definicji wskaźnika należy podać rozmiar tablicy**, chociaż definicja ta nie rezerwuje pamięci na tablicę; pamięć jest tylko rezerwowana na wskaźnik *P*. Podanie rozmiaru tablicy (tu 8) definiuje typ wskaźnika, a tym samym wielkość wskazywanego obszaru i arytmetykę na tym wskaźniku (przesunięcie wskazania przy inkrementacji wskaźnika). Wyrażenie *P*+1 wskaże tu o osiem liczb typu *float* dalej, niż wskazuje *P*.

Tablice znakowe służące do przechowywania tekstów muszą pomieścić niewidoczny znak końca tekstu \0.

Na przykład, aby zapamiętać 15-znakowy tekst w tablicy *Z*1 oraz aby zapamiętać w tablicy *Z*2 12 tekstów każdy o długości do 23 znaków należy zdefiniować

```
char Z1[16], Z2[12][24];
```

Pytania i zadania

2.12. Zdefiniować tablice współczynników wielomianów
$A(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$
$B(x) = b_m x^m + b_{m-1}x^{m-1} + ... + b_1 x + b_0$
$C(x) = A(x) \cdot B(x)$.
Zdefiniować stałe N, M i założyć $n < N$ oraz $m < M$.

2.13. Zdefiniować tablice *A, B, C* postaci:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1k} \\ b_{21} & b_{22} & \cdots & b_{2k} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nk} \end{bmatrix}, \quad \text{gdzie } C = A \times B,$$

$$c_{ij} = \sum_{l=1}^{n} a_{il} b_{lj}.$$

2.14. Zdefiniować tablice *A*, *B*, które posłużą do zapamiętywania układu N równań liniowych z N niewiadomymi oraz tablicę X na rozwiązania tego układu. Tablica A zawiera współczynniki a_{ij} (przy j-tej niewiadomej w i-tym równaniu). Tablica B zawiera wyrazy wolne b_i. Tablica X zawiera rozwiązania x_j.

2.15. Zdefiniować tablicę tekstów do zapamiętywania
a) nazw dni tygodnia, b) nazw miesięcy, c) nazw pór roku.

2.16. Jaki atrybut klasy pamięci powoduje, że definiowane zmienne
a) istnieją podczas wykonywania całego programu,
b) są w miarę możliwości umieszczane w rejestrach procesora,
c) nie otrzymują przydzielonej pamięci.

2.6. Struktury i unie

Struktura jest obiektem złożonym z jednej lub kilku, zazwyczaj różnych typów, zmiennych nazywanych polami i dla wygody zgrupowanych pod jedną nazwą. Struktura odpowiada rekordowi w języku Pascal. W odróżnieniu od tablicy elementy (pola) struktury mogą być różnego typu.

Definicja struktury ma postać

```
struct NAZWA {
        definicja pola
        . . . . . .
        definicja pola
       } lista identyfikatorów ;
```

Definicja pola ma postać zwykłej definicji bez specyfikacji klasy pamięci i może definiować kilka pól na raz.

Nazwa struktury (*NAZWA*) oraz lista identyfikatorów są opcjonalne. Jeśli brak jest listy identyfikatorów, to deklaracja tylko opisuje strukturę, nie definiując ani nie zapowiadając żadnych obiektów. Obiekty te można później definiować powołując się już tylko na nazwę struktury.

Przykłady definicji i deklaracji struktur

```
struct  Osoba {
   char imie[16], nazwisko[24];
                              // pola znakowe typu char[16] i char[24]
   short wiek, wzrost;        // pola typu short
  } P1,P2;                    // P1, P2 są zdefiniowanymi strukturami

struct  FIGURA {
   char nazwa[12];            // pole typu char[12]
   float obwod, pole;         // 2 pola typu float
  };                          // koniec opisu (specyfikacji) struktury

struct  FIGURA A1, *pA;       // definicja struktury A1 i wskaźnika pA
```

Do pól struktury można odwołać się bezpośrednio operatorem **.** (kropka) lub przez wskaźnik operatorem –> (minus, większe). Na przykład

```
A1.obwod
pA->obwod
```

są odwołaniami do pola *obwod*. Obu tych wyrażeń można używać tak, jak używa się identyfikatorów zmiennych typu *float*. Na przykład

```
A1.obwod = 17.5;
```

podstawia wartość 17.5 do pola *obwod* struktury *A*1,

```
pA->obwod = 20.3;
```

podstawia 20.3 do pola *obwod* struktury wskazywanej przez *pA*.

Wyrażenie

```
P1.imie
```

można używać tak jak nazwę tablicy znakowej typu *char*[12], natomiast wyrażenie

```
P1.imie[0]
```

daje w wyniku kod (ASCII) pierwszej litery (znaku) imienia. Wyrażenia tego można używać tak jak zmiennej typu *char*.

Unia jest odmianą struktury. Pola w strukturze są umieszczone w pamięci jedno za drugim, natomiast pola w unii są umieszczone jedno na drugim począwszy od tego samego miejsca w pamięci. O ile więc struktura przechowuje równocześnie wartości wszystkich swoich pól, to unia w danej chwili przechowuje wartość tego pola, do którego zapisu dokonano najpóźniej.

Unie definiuje się podobnie jak struktury, z tą różnicą, że w miejscu słowa kluczowego *struct* występuje słowo *union*. Np.

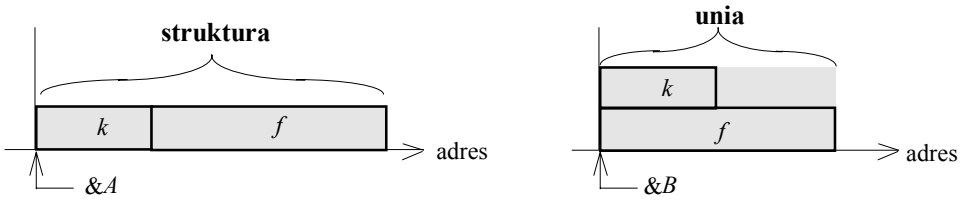
```
union {
        char t[10];
        int i;
        long l;
        double a;
   }  T;
```

Zasady dostępu do pól unii są takie same jak do pól struktury za pomocą operatorów `.` oraz `->`. Różnicę między organizacją pamięci struktury a unii o takich samych składowych pokazano na rys. 2.1:

```
struct {
    int k;
    float f;
  } A;
```

```
union {
    int k;
    float f;
  } B;
```

Jak widać składowe *k* i *f* struktury *A* są umieszczone jedna za drugą i zajmują odrębne obszary pamięci, natomiast składowe unii *B* są umieszczone jedna na drugiej, zajmując wspólny obszar pamięci.



Rys. 2.1. Organizacja struktury *A* i unii *B* w pamięci

Identyfikatory etykiet, identyfikatory składowych struktur, klas i unii oraz identyfikatory typów i zmiennych tworzą trzy klasy, które wzajemnie się nie przesłaniają.

W przykładzie podanym poniżej zmienna *K* i składowa *K* nie przesłaniają się

```
{
  struct PP
    { int K;
      char *p;
    } K;
  K.K=1;
  . . . . . .
}
```

Pytania i zadania

2.17. Podaći ile co najmniej bajtów w implementacji Borland C++ musi zajmować struktura:

```
a) struct {          b) struct {          c) struct {
     char W[7];           short  D[5];         char  a,b,c,d;
     int  k[4];           double dl;           int   i,j,k;
     long y;};            float  a[2];};     };
```

Ile bajtów zajmowałaby unia o takich samych polach?

2.18. Napisać specyfikacje struktur do zapamiętywania
a) adresu zamieszkania,
b) imienia, nazwiska ucznia i jego średnich ocen z przedmiotów,
c) daty z nazwą miesiąca i dnia tygodnia.

2.19. Co otrzyma się w wyniku odczytu pola *T* unii

```
union {char  T[12]; int  K; long  L; double X; }
```

jeśli zapisano pole a) *K*, b) *L*, c) *X*?
Które elementy tablicy *T* warto odczytywać?

2.7. Inicjowanie

Deklarator może określać wartość początkową definiowanego identyfikatora. Inicjator składa się ze znaku = i z wyrażenia albo ujętej w nawiasy { } listy wyrażeń. Na przykład

```
int k=1, j=15, L[5]={3,2,1};
double y=0.5;
```

definiuje zmienne *k, j*, nadając im wartości 1 i 15 oraz definiuje pięcioelementową tablicę, inicjując ją wartościami: $L[0]=3$, $L[1]=2$, $L[2]=1$, $L[3]=L[4]=0$. Zmiennej *y* nadaje się wartość 0.5.

Inicjowanie odbywa się zgodnie z zasadami:

- Zmienne statyczne lub zewnętrzne są inicjowane wyrażeniami stałymi raz przed rozpoczęciem programu.
- Zmienne statyczne lub zewnętrzne nie zainicjowane otrzymują wartości zerowe.
- Zmienne automatyczne i rejestrowe otrzymują inicjowane wartości za każdym razem podczas ich tworzenia w chwili napotkania definicji w procesie wykonywania programu.
- Można inicjować tablice statyczne lub zewnętrzne.
- Lista inicjacyjna tablicy nie może zawierać więcej wyrażeń niż tablica ma elementów. Jeśli lista zawiera mniej wyrażeń, to pozostałe elementy są inicjowane zerami.
- Jeśli inicjowanym elementem tablicy jest inna tablica, to lista wyrażeń inicjacyjnych dla tego elementu może być ujęta w nawiasy klamrowe { } i nie musi być wtedy kompletna. Pozostałe składowe tego elementu będą inicjowane zerami.
- W definicji tablicy z listą inicjacyjną można pominąć rozmiar tablicy. Zostaje wtedy przyjęty taki minimalny rozmiar, aby tablica pomieściła wszystkie elementy listy inicjacyjnej.
- Tablice i wskaźniki znakowe można inicjować tekstem.

Przykłady

Przyjmijmy, że wewnątrz pewnej funkcji zmienne *N*, *M* oraz *k* zdefiniowano następująco:

```
static int N, M=5;
int k=2;
```

Zaraz po uruchomieniu programu zmienne *N* oraz *M* otrzymają wartości $N=0$, $M=5$. Zmienna *k* będzie otrzymywać wartość $k=2$ po każdorazowym wywołaniu tej funkcji. Zmienne *N, M* będą miały wartości takie jak w chwili powrotu z poprzedniego wywołania tej funkcji.
Definicja

```
float X[8]={1,3,5);
```

nadaje wartości: $X[0]=1$, $X[1]=3$, $X[2]=5$, $X[3]=X[4]=X[5]=X[6]=X[7]=0$.
Definicje:

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

są sobie równoważne i nadają elementom tablicy *K* wartości

$$K = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 \\ 3 & 4 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Należy jednak unikać opuszczania nawiasów w listach inicjacyjnych, ponieważ łatwo wtedy o pomyłkę. Kompilator może ostrzegać o braku nawiasów w liście inicjacyjnej. Gdyby rozmiar powyższej tablicy nie był podany, to definicje

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

definiowałyby tablicę 2-wierszową

$$K = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 \\ 3 & 4 & 5 & 0 & 0 \end{bmatrix}$$

Definicje:

```
int A[3][3]={1,2,3,4};
int B[3][3]={{1},{2},{3},{4}};
int C[ ][3]={1,2,3,4};
int D[ ][3]={{1},{2},{3},{4}};
```

definiują trzy różne tablice *A, C, D*. Definicja tablicy *B* jest błędna, ponieważ lista inicjacyjna zawiera wartości dla czterech wierszy, podczas gdy tablica posiada ich tylko trzy.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \qquad C = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 0 \end{bmatrix}, \qquad D = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 0 \\ 4 & 0 & 0 \end{bmatrix}.$$

Definicja

```
char T[]="Borland C++";
```

definiuje 12-elementową tablicę T zainicjowaną tekstem „Borland C++”. Ostatni element tablicy *T*[11] zawiera znak zerowy.

Struktury inicjuje się pole po polu, np.

```
struct FIGURA A1={"kwadrat", 20, 25};
```

inicjuje pola: *A*1.*nazwa* = "kwadrat", *A*1.*obwod* = 20 oraz *A*1.*pole* = 25.

```
struct RZYM
  { char *symbol;
    int wartosc; }
      RR[ ]={{"M",1000},{"CM",900},{"D",500},{"CD",400},
          {"C",100},{"XC",90},{"L",50},{"XL",40},{"X",10},
          {"IX",9},{"V",5},{"IV",4},{"I",1}};
```

definiuje i inicjuje tablicę 13 struktur nadając polom *tekst* wskazania do tekstów składowych liczb rzymskich a polom *wartosc* – odpowiadające tym tekstom wartości. Na przykład *RR*[2].*wartosc* = 500, a *RR*[2].*symbol* wskazuje na tekst "D".

Powyższa lista inicjacyjna zawiera komplet wartości, można zatem pominąć wewnętrzne nawiasy ujmujące wartości dla poszczególnych struktur, co jednak może wywołać ostrzeżenie kompilatora. Elementy tablicy *RR* typu *struct RZYM* zawierają wskaźniki do umieszczonych gdzieś w pamięci tekstów z listy inicjacyjnej. Jest to zalecane, gdy teksty znacznie się różnią długością, lub gdy długości tekstów są trudne do przewidzenia. W tym przypadku mamy teksty jedno i dwuznakowe i ekonomiczniej jest umieścić je bezpośrednio w strukturach, zastępując wskaźnik trzybajtową tablicą znakową. Definicja takiej struktury wraz z listą inicjacyjną tablicy *RR* z częściowo opuszczonymi nawiasami może być następująca

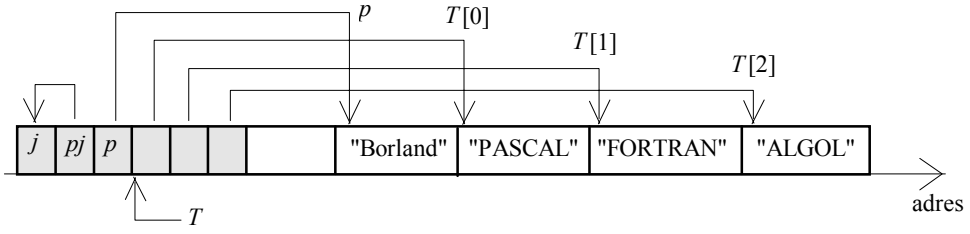
```
struct RZYM
  { char symbol[3];
    int wartosc; }
      RR[ ]={"M",1000,"CM",900,"D",500,"CD",400,"C",100,
             "XC",90,"L",50,"XL",40,"X",10,"IX",9,"V",5,
             "IV",4,"I",1};
```

Zmienne wskazujące inicjuje się podobnie, np.

```
int  j, *pj=&j;
char *p="Borland",
       *T[ ]={"PASCAL","FORTRAN","ALGOL"};
```

zdefiniowano zmienną wskazującą *pj* i nadano jej wskazanie do wcześniej zdefiniowanej zmiennej *j*. Zmienna *p* wskazuje na tekst *Borland*. Tablica *T* zawiera trzy wskaźniki do trzech tekstów.

Na rys. 2.2 pokazano rezerwację pamięci dokonaną przez podane definicje zmiennej *j*, zmiennych wskaźnikowych *pj*, *p* oraz tablicy trzech wskaźników *T*. Definicje te rezerwują też odpowiednie obszary pamięci na inicjowane teksty i nadają wskaźnikom wartości wskazań na te teksty.



Rys. 2.2. Rozmieszczenie zmiennych *j*, *pj*, *p* oraz tablicy *T* w pamięci

Częste błędy w definiowaniu i inicjowaniu tablic to:
- Zapominanie o znaku końca tekstu '\0' przy definiowaniu tablic tekstowych.
- Użycie indeksu N (np. $A[N]$) do N -elementowej tablicy (np. zdefiniowanej jako *int* $A[N]$;) zapominając, że ostatni element ma numer $N-1$.
- Definiowanie tablicy bez podania jej rozmiaru i bez listy inicjacyjnej.
- Inicjowanie tablic automatycznych, na co czasami pozwala kompilator.

Pytania i zadania

2.20. Zdefiniować tablice zainicjowane:
a) liczbami dni w każdym miesiącu roku zwykłego i przestępnego,
b) nominałami złotówkowymi będących w obiegu banknotów.

2.21. Jakie wartości otrzymają elementy tablic
a) `int a[10]={5}, b[3][3]={{1, 2}, {3}, 4};`
b) `double x[3][5]={{0, 0.5, -0.5}, {1.5, 2}};`
c) `int y[5][2]={{1, 2},{ 3}, {4}, 0, 5, 6};`
d) `long L[2][2][2]= {{{0}, {11, 12}}, {20}};`

2.22. Zdefiniować i zainicjować tablice do konwersji liczb dziesiętnych na rzymskie. Jedna tablica powinna zawierać teksty symboli, z których buduje się liczby rzymskie, druga tablica zaś powinna zawierać wartości związane z tymi symbolami.

2.23. Zdefiniować tablice *A, B, C* o elementach typu *int* i zainicjować je

$$A=\begin{bmatrix}0&0&1&2\\0&3&4&0\\5&6&0&0\end{bmatrix},\qquad B=\begin{bmatrix}5&6&7\\8&0&0\\0&9&0\\0&0&0\end{bmatrix},\qquad C=\begin{bmatrix}1&2&3&0&0\\4&5&0&6&0\\7&8&0&0&0\\9&4&0&0&0\end{bmatrix}.$$

2.24. Jakie będą wartości i wymiary tablic
a) `double a1[][3]={1,0,2,3,4}, a2[]={1,0,2,3,4};`
b) `int b1[][2]={{1,2},{3},4}, b2[][3]={{1},{2,3},4};`
c) `long c[][2][2]={{{1,2},{3,4}},{{5}},{{6},{7}},{{8}},9,10,11};`
d) `char d[]="Ekran monitora";`
e) `char e[]="Jeden\n Dwa\n Trzy\n";`
f) `char f[]="Jeden\n""Dwa\n""Trzy\n";`
g) `char g[][6]={"Jeden","Dwa","Trzy"};`

2.25. Jakie wartości otrzymają pola struktury *S* po zainicjowaniu

```
struct {    char miesiac [12];
            int dzien;} S =  {"Luty",10};
```

2.8. Identyfikatory typów

Jeśli deklaracja zaczyna się specyfikatorem **typedef**, to deklarowane identyfikatory są nazwami typów.

Na przykład

```
typedef int VEKTOR[20], TABLICA[12][30], *WSKAZNIK;
```

VEKTOR jest identyfikatorem typu *int*[20], *TABLICA* – typu *int*[12][30] a *WSKAZNIK* – *int**. Dalej można tych identyfikatorów używać tak jak nazw typów. Poniższe definicje są na przykład parami równoważne:

```
VEKTOR  A,B[120];          int  A[20], B[120][20];
TABLICA  C,D[8];           int  C[12][30], D[8][12][30];
WSKAZNIK  F,G[10];         int  *F, *G[10];
```

O ile definicje obiektów *A, C, F* są oczywiste, to interpretacja obiektów *B, D* oraz *G* może nasuwać wątpliwości. Zauważmy, że np. *B* jest tablicą 120 elementów typu *VEKTOR*, czyli tablic 20 liczb całkowitych typu *int*. Tablica *B* zawiera więc 120 wierszy po 20 liczb w każdym. Podobnie tablica *D* zawiera osiem tablic o wymiarach 12×30, tablica *G* zaś zawiera dziesięć wskaźników (elementów typu *WSKAZNIK*).

Są dwa główne powody używania deklaracji *typedef*.

1. **Parametryzacja programu**. Gdy w dużym programie w wielu punktach deklarowano zmienne typu *VECTOR*, *TABLICA* i *WSKAZNIK*, to zmiana tych typów (np. zmiana rozmiarów wektorów i tablic), wymaga tylko zmiany deklaracji typów. Bez tych deklaracji należałoby zmieniać program w wielu miejscach, wszędzie tam, gdzie występują definicje obiektów modyfikowanych typów.
2. **Poprawa czytelności programu**. Identyfikatorem typu może być takie słowo, które ten typ opisuje, jak uczyniono w ostatnim przykładzie.

Pytania i zadania

2.26. Co oznacza definicja (podać definicję równoważną)

```
AA  a[5],b,c[2][3];
```

jeśli wcześniej zdefiniowano *AA*

a) *typedef int AA*;
b) *typedef float AA*[100];
c) *typedef static char AA*[60];
d) *typedef long AA*[8][16];
e) *typedef char *AA*;
f) *typedef double *AA*[24];

2.27. Zdefiniowano typy **`typedef int BB[25], CC[40][15];`**
Użyj tych typów do zdefiniowania zmiennych
a) *int A*[25], *B*[15][25], *C*[7][40][15];
b) *int K*[25][40][15], *M*[40][15][25];
c) *extern int N*[40][15];

2.28. Zdefiniuj odpowiednie nazwy typów, które oznaczają:
a) wskaźnik do nazwy dnia tygodnia,
b) tablicę nazw miesięcy,
c) tablicę zawierającą daty w postaci: rok oraz numery miesiąca i dnia,
d) tablicę zawierającą współrzędne (*x, y*) punktów na płaszczyźnie oraz tablicę sto takich punktów.

2.9. Zmienne wyliczeniowe

Specyfikator wyliczenia ma postać

enum TYP {wyliczenie, wyliczenie, ..., wyliczenie};

w którym słowo *TYP* jest identyfikatorem typu wyliczeniowego i może być opuszczone. Każde wyliczenie ma jedną z dwu postaci:

identyfikator
identyfikator = inicjator

Na przykład

```
enum Boolean {FALSE, TRUE};
enum  {FALSE, TRUE};
enum text_modes
       {LASTMODE=-1, BW40=0, C40, BW80, C80, MONO=7};
```

Identyfikatorom wyliczenia (FALSE, TRUE, LASTMODE, BW40, C40, BW80, C80, MONO) przypisuje się wartości całkowite podane w inicjatorze.

Jeśli inicjator nie występuje, to identyfikatorowi przypisuje się wartość o 1 większą niż poprzedniemu. Dla pierwszego identyfikatora domniemywa się wartość 0. Identyfikatory wyliczeń oznaczają stałe całkowite.

W przytoczonych przykładach FALSE=0, TRUE=1, C40=1, BW80=2, C80=3. Można definiować zmienne wyliczeniowe. Na przykład definicje

```
enum Boolean  L1, L2;
enum  {FALSE, TRUE} L1, L2;
```

definiują zmienne wyliczeniowe *L*1 i *L*2, którym można nadawać wartości określone w wyliczeniu.

Posługiwanie się identyfikatorami jest często wygodniejsze niż liczbami. Wygodniej jest np. posługiwać się nazwami kolorów niż odpowiadającymi im liczbami. Program jest wtedy bardziej zrozumiały. Głównym powodem używania zmiennych wyliczeniowych jest to, że kompilator ostrzega o użyciu wartości nie należącej do danego typu wyliczeniowego. Na przykład przypisanie *L*1 = 5; wywoła ostrzeżenie kompilatora. Poza tym zmienne typu wyliczeniowego są traktowane tak jak zwykłe zmienne całkowite.

Pytania i zadania

2.29. Jakie wartości nadano identyfikatorom w wyliczeniach
   a) *enum* {SOLID, DOTTED, CENTER, DASHED, USERBIT};
   b) *enum* {LEFT, CENTER, RIGHT, BOTTOM = 0, TOP = 2};
   c) *enum* {CGALO, CGAC1, CGAC2, CGAC3, CGAHI, EGALO = 0, EGAHI, EGAMONOHI, HERCMONOHI = 0, VGALO = 0, VGAMED, VGAHI};

2.30. Napisz wyliczenia
   a) kolorów w języku polskim oraz atrybutu migania,
   b) kodów polskich liter w systemie Windows: ć–134, ł–136, Ź–141, Ć–143, Ś–151, ś–152, Ł–157, ó–162, Ą–164, ą–165, Ę–168, ę–169, ź–171, Ż–189, ż–190, Ó–224, Ń–227, ń–228.

Zadania laboratoryjne

2.31. Napisz program, który wypisze rozmiary kilku wybranych typów.

2.32. Napisz program, który obok nazwy każdego miesiąca wypisze ile dni upłynęło od początku roku do końca tego miesiąca. W programie zainicjuj tablicę nazw miesięcy i tablicę dni w miesiącach.

2.33. Napisz program z funkcją, która będzie pamiętać ile razy była wywoływana i przy każdym wywołaniu wyprowadzi na ekran numer wywołania.

2.34. Napisz program w dwu plikach. W jednym pliku zdefiniuj i zainicjuj zmienną. W drugim pliku funkcja *main* wydrukuje wartość tej zmiennej. Zapoznaj się z kompilacją i uruchamianiem programów napisanych w kilku plikach.

2.35. Zdefiniuj typ wyliczeniowy i zmienną tego typu. Podejmij próby nadawania tej zmiennej wartości, które nie są przewidziane dla zmiennych tego typu. Czy próby zakończą się sukcesem?

3. Wyrażenia i operatory

Wyrażenie jest konstrukcją językową określającą operacje, które mają być wykonane w celu obliczenia wartości i jej typu. Wykonanie tych operacji nazywa się **opracowaniem wyrażenia**. Opracowanie wyrażenia składa się z prostych operacji 1-, 2- lub 3-argumentowych, takich jak np.: negacja, mnożenie, operator wyboru.

O kolejności wykonywania operacji decydują priorytety operatorów.

Jeśli priorytety operatorów są jednakowe, to o kolejności decyduje wiązanie. Wiązanie lewe nakazuje wykonanie najpierw operatora z lewej strony, a wiązanie prawe nakazuje wykonanie operatora z prawej strony.

Na przykład, mnożenie będzie wykonywane przed dodawaniem, a dodawanie przed przypisaniem. Wyrażenie *A–B–C* będzie opracowywane jako (*A–B*)–*C*, ponieważ operator odejmowania ma wiązanie lewe.

Argumenty z operatorów łącznych i przemiennych (*, +, &, |, ^) mogą być opracowywane w dowolnej kolejności.
Nie określa się kolejności opracowywania wyrażeń, które są argumentami funkcji.

Na przykład $A+(B+C)$ może być opracowane jako $(A+B)+C$. Wyrażenie $A+(B+(A++))$ nie jest poprawne, ponieważ może dać w wyniku 2A+B lub 2A+B+1.
Wyrażenie *Fun*($x=a+b$, $x+y$) nie jest poprawne, ponieważ nie wiadomo, czy najpierw będzie opracowane $x+y$ czy $x=a+b$. Wartość wyrażenia $x+y$ nie jest zatem jednoznacznie określona.

Jeśli operator zmienia wartość swojego argumentu, to ten argument musi być *l*-wartością (*l*-value). Z definicji *l*-wartością jest to, co może wystąpić po lewej stronie operatora przypisania. Nazwa zmiennej jest *l*-wartością. Niektóre operatory dają w wyniku *l*-wartość. Np. jeśli *px* jest wskaźnikiem na *l*-wartość, to **px* jest tą *l*-wartością, odnoszącą się do wskazywanego obiektu. Tak więc jeśli zdefiniujemy np. *float x, *px=&x;*, to wyrażenie **px* jest równoważne zmiennej *x* i może wystąpić wszędzie tam, gdzie może wystąpić zmienna *x*.

Tabela 3.1. Operatory

Grupa	Priorytet	Wiązanie	Operator	Uwagi (*) – brak w standarcie języka
Kwalifikatory	0	prawe	**::** ***Klasa*::**	*globalność, zakres* (*)
Tablice, struktury, funkcje i operatory jednoargumentowe	1	lewe	**[ ] . –> ( )** *Typ***( )** **++ --**	indeksowanie, wybór, wywołanie *konwersja funkcyjna* (*) in- i dekrementacja następnikowa
	2	prawe	**++ --** **sizeof ! ~ & * (***Typ***) – + new delete**	in- i dekrementacja poprzednikowa rozmiar, negacja logiczna i bitowa adresacja, wyłuskanie, rzutowanie, minus, plus*, rezerwacja, zwolnienie* (*)
Komponenty	3	lewe	**–>* .***	*wybór komponentów* (*)
Arytmetyczne	4	lewe	*** / %**	mnożenie, dzielenie, reszta
	5	lewe	**+ –**	dodawanie, odejmowanie
Przesuwania	6	lewe	**<< >>**	przesuwanie w lewo i w prawo
Porównania	7	lewe	**< <= > >=**	nierówności ostre i nieostre
	8	lewe	**= = !=**	równe, różne
Bitowe	9	lewe	**&**	iloczyn bitowy AND
	10	lewe	**^**	różnica symetryczna EX-OR
	11	lewe	**|**	suma bitowa OR
Logiczne	12	lewe	**&&**	koniunkcja (and)
	13	lewe	**||**	alternatywa (or)
Wyboru	14	prawe	**? :**	3-argumentowy operator wyboru
Przypisania	15	prawe	**= *= /= %= += –= <<= >>= &= ^= |=**	przypisanie oraz przypisania z automodyfikacją
Połączenia	16	lewe	**,**	operator połączenia

Konwersje standardowe:
1. Argumenty typu *char* i *short* są przekształcane do typu *int*.
2. Jeśli wieloargumentowy operator wymaga, by jego argumenty były jednakowego typu, to argument typu węższego jest przekształcany do typu szerszego. Typem szerszym może być w kolejności od węższego do najszerszego: *unsigned, long, unsigned long, double, long double*.

Wszystkie opracowane wyrażenia dają wynikowe wartości określonego typu. Na przykład wynikiem wyrażenia $A+B$ jest wartość sumy, a wyrażenia $A=B$ jest wartość B po konwersji do typu zmiennej A. Efektem ubocznym jest tu przypisanie zmiennej A wartości wynikowej.

Argumenty operatorów są przed wykonaniem operacji poddawane konwersji standardowej. Na przykład w dodawaniu dwu argumentów typu *char* i *float* ich typy są następująco przekształcane

char + *float* jest przekształcane w pierwszym etapie do *int* + *float*
a następnie w drugim etapie do *float* + *float*.

W tabeli przedstawiono operatory zgrupowane na kolejnych poziomach priorytetów w kolejności od najwyższego. Tak więc najwyższy priorytet 0 mają kwalifikatory globalności i zakresu (:: *Class*::) a najniższy priorytet 16 ma operator połączenia (,). Tabela zawiera też operatory kompilatora obiektowego Borland C++.

3.1. Kwalifikator globalności

::*Identyfikator_globalny*

Wynikiem jest przesłonięta zmienna globalna, nazwa funkcji albo nazwa typu globalnego.

Przykład:

```
#include <stdio.h>
int K=10;
void main(void)
  { int K=1;
    printf("%d",K+::K);
      { int K=3;
        printf("   %d",K+::K);}
  }
```

Program wyprowadzi liczby 11 13.

3.2. Operatory indeksowania, wyboru i wywołania

Operator indeksowania

***Tablica* [*Indeks*]**

Jedno z wyrażeń (*Tablica, Indeks*) musi być typu wskazującego (najczęściej *Tablica* – nazwa tablicy jest stałą wskazującą na początkowy element tablicy), a drugie musi być typu całkowitego (najczęściej *Indeks*). Wynikiem jest zmienna wskazywana przez wyrażenie *Tablica* + *Indeks*. Operator ten jest przemienny.

Na przykład po zdefiniowaniu

```
double T[80], A[120];
T[i],  *(T+i),  *(i+T),  i[T]  // są równe i-temu elementowi tablicy T
A[8]=3;                        // można zapisać w postaci 8[A]=3;
```

Wyznaczanie elementów tablic wieloindeksowych może być wykonywane etapami

```
float B[40][50], *p;
for(i=0; i<40; i++)
  { p=B[i];                          // obliczenie wskazania do wiersza
    for(j=0; j<50; j++) p[j]=5.0;    // równoważne B[i][j]=5.0
  }
```

Operatory wyboru

Struktura . Pole
Wskaźnik_Struktury –> Pole

Wynikiem jest nazwa pola *Pole* danej lub wskazywanej struktury.

Przykład:

```
struct PUNKT {
    char Nazwa[8];
    float x, y;
  }  P1={"PA",1,3}, *WP1=&P1;        // WP1 wskazuje na P1
```

Poniższe wyrażenia dają parami jednakowe wyniki:

Wyrażenie z operatorem		Wynik
`.`	`->`	
`P1.Nazwa`	`WP1->Nazwa`	wskazanie ośmioznakowej tablicy
`P1.Nazwa[1]`	`WP1->Nazwa[1]`	element tablicy z literą *A*
`P1.x`	`WP1->x`	zmienna typu *double* o wartości 1
`P1.y`	`WP1->y`	zmienna typu *double* o wartości 3

```
struct KOLO {
    struct PUNKT S;
    float R;
  } K1, *pK1=&K1;                    // pK1 wskazuje na K1
```

Przykłady par równoważnych podstawień

```
K1.R=50     albo    pK1->R=50
K1.S.x=2            pK1->S.x=2
K1.S.y=6            pK1->S.y=6
```

Wartością wyrażeń

```
K1.S        oraz    pK1->S
```

jest struktura typu *struct PUNKT*. Następny operator *kropka* wybiera pole *x* lub *y* tej struktury.

Operator wywołania funkcji

Funkcja **(** ***Arg*****1,** ***Arg*****2, ...,** ***Argn*** **)**

Nazwa funkcji jest wyrażeniem wskazującym funkcję, a każde *Arg* jest aktualnym argumentem wywołania funkcji. Wynikiem jest stała zwracana przez funkcję instrukcją *return*. Funkcje typu *void* nie zwracają wartości (podobnie jak procedury w języku PASCAL).

Przykład

```
#include<stdio.h>
double Pole(double r)
   {return 3.1416*r*r;
   }
void main(void)
   {printf("%lf",Pole(10.0));
   }
```

Taki sam efekt można osiągnąć, jeżeli wywoła się funkcję poprzez wskaźnik, np.

```
void main(void)
 { double (*f)(double);
   f=Pole;
   printf("%lf", f(10.0) );
 }
```

Wynikiem funkcji *pole* jest pole koła o promieniu *r*. Program wyprowadzi liczbę 314.16 .

Pytania i zadania

3.1. Zapisz wyrażenia używając operatora indeksowania [ ], zakładając, że *x, y, z* są wskaźnikami, a *i*, *j*, *k* są typu *int*.

a) `*(x+5)`	b) `*z`	c) `**y`
d) `*(x+i)`	e) `**(y+j)`	f) `x+i`
g) `*(x+i+j)`	h) `*(*y+k)`	i) `*(*(y+i)+j)`

3.2. Zapisz wyrażenia bez użycia operatora indeksowania [ ]

a) `C[0]`	b) `*C[j]`	c) `c) B[i-j]`
d) `&B[k]`	e) `A[0][0]`	f) `f) A[0][1]`
g) `&A[k][0]`	h) `A[i][j]`	i) `i) A[i-k][j]`

3.3. Nadaj przykładowe wartości polom struktury, które zdefiniowano

```
struct PUNKT{
   char NAZWA[2];
   float x, y;}  A;
struct Odcinek{
   struct PUNKT *p1, P2;
 }  S, *pS=&S;
```

a) posługując się nazwą struktury *S*,

b) posługując się wskaźnikiem *pS* do struktury *S*.

3.3. Operatory jednoargumentowe

Operatory inkrementacji i dekrementacji

++ Var ***Var ++***
–– Var ***Var ––***

Var jest *l*-wartością typu arytmetycznego lub wskazującego. Efektem ubocznym jest zwiększenie (dla operacji ++) lub zmniejszenie (dla operacji --) *Var* o 1. Wynikiem jest wartość *Var* po operacji poprzednikowej lub przed operacją następnikową. Typ wyniku taki jak typ *Var*.

Priorytet	Operacja	Efekt uboczny	Wynik
1	*Var*++	*Var* = *Var* + 1	Wartość *Var*
	Var––	*Var* = *Var* – 1	przed wykonaniem
2	++*Var*	*Var* = *Var* + 1	Wartość *Var*
	––*Var*	*Var* = *Var* – 1	po wykonaniu

Przykłady

```
int j=1, k=1, A[5];
A[++j]=7;          // A[2]=7,    j=2
A[k++]=10;         // A[1]=10,   k=2
j=++k+k++;         // j=6,       k=4
```

Zauważmy, że wyrażenia: ++*k*++ oraz *k*++++ nie są poprawne, ponieważ wynikiem *k*++ jest stała, a nie *l*-wartość.

Użycie inkrementacji lub dekrementacji następnikowej (*Var*++, *Var*––) tam, gdzie powinna być poprzednikowa (++*Var*, ––*Var*) lub odwrotnie jest częstym błędem.
Złym nawykiem jest używanie podstawienia np. *k*=*k*+1; tam, gdzie można użyć inkrementacji lub dekrementacji.

Operator rozmiaru

sizeof(***Typ*** **)**
sizeof ***Wyrażenie***

Wynikiem jest stała typu *int* o wartości równej liczbie bajtów zajmowanych przez daną typu *Typ* lub typu takiego jak wynik wyrażenia *Wyrażenie*. Jeśli wyrażeniem jest nazwa tablicy, to wartość wyniku jest równa rozmiarowi całej tablicy w bajtach.

Przykłady

```
char BUF[sizeof(double)];     // BUF pomieści zmienną typu double
float A[4][5];
printf("%d", sizeof A );      // sizeof A==20 sizeof(float)
sizeof(A[0][1]+A[1][1])       // sizeof(float)
```

Stosowanie wartości liczbowych zamiast operatora *sizeof* lub definicji *#define* jest złym nawykiem, który prowadzi do pisania programów słabo sparametryzowanych. Programy takie trudno przenosi się na inne maszyny i trudno się je modyfikuje.

Operator zaprzeczenia logicznego (not)

! ***Wyrażenie***

Wyrażenie może być typu arytmetycznego albo wskazującego. Wynik jest stałą typu *int* o wartości 0 albo 1.

$$!a = \begin{cases} 0 & \text{gdy} \quad a \neq 0 \\ 1 & \text{gdy} \quad a = 0 \end{cases}$$

Przykłady

```
!0 = 1    !0.0 = 1          !5 = 0      !'K' = 0
!'0' = 0  !'\0' = 1         !2.5 = 0    !!2.5 = 1
```

Operator zanegowania bitów (NOT)

~ ***Wyrażenie_całkowite***

Jeśli wyrażenie jest typu *char* lub *short*, to zostaje ono przekształcone do typu *int*, a następnie wszystkie jego bity zostaną zamienione na przeciwne. Wynikiem jest stała całkowita typu podanego wyrażenia po konwersji wstępnej.

Przykłady

```
 i = 00111000 00001101 = 14349
~i = 11000111 11110010 = 51186 (–14350, gdy i jest typu int)
 i = 00000000 00000000 = 0
~i = 11111111 11111111 = 65535 (–1, gdy i jest typu int)
```

Operator zmiany znaku (minus)

– Wyrażenie_arytmetyczne

Wynikiem jest stała o wartości przeciwnej i typie po konwersji wstępnej.

Przykłady

Przyjmijmy, że zmienne *c* oraz *x* zdefiniowano następująco:

```
char c = 'A';   float x = 2.5;
```

`-c` daje –65 typu *int*,
`-x` daje –2.5 typu *double*,
`- -x` daje 2.5 typu *double* (spacja pomiędzy minusami),
`--x` daje 1.5 typu *double* (dekrementacja `--`),
`-3.5f` daje –3.5 typu *double* (operator *minus* i stała typu *float*).

Operatory adresacji (&) i wyłuskania (*)

& Nazwa
**** Wyrażenie_wskazujące***

Nazwa i wyrażenie wskazujące mogą dotyczyć zmiennej albo funkcji.

Wynikiem *&Nazwa* jest stała wskazująca podaną zmienną albo funkcję. Jeśli typem *Nazwa* jest *Typ*, to *&Nazwa* jest typu *Typ**.

Wynikiem **Wyrażenie_wskazujące* jest nazwa wskazywanej zmiennej albo funkcji. Jeśli typem wyrażenia jest *Typ**, to **Wyrażenie_wskazujące* jest typu *Typ*.

Przykłady

```
double x, *px;
px = &x;                // do px podstawia się wskazanie zmiennej x
*px = 3.7;              // zmienna x otrzymuje wartość 3.7
```

W powyższym przykładzie wynikiem wyrażenia **px* jest zmienna *x*, gdyż *px* wskazuje na *x* (przypisano *px=&x*;).

Operator rzutowania (konwersji)

(Typ) Wyrażenie

Wynikiem jest stała podanego typu o wartości wyrażenia przekonwertowanej do podanego typu.

Przykłady

```
float x=2.9;
int K[2]={0x4241, 0x43};
char z, *p;
printf("%d", (int)x );          // wyprowadzi liczbę 2
printf("%s", (char*)K);         // wyprowadzi napis ABC
z = *((char*)K+1);              // z otrzymuje kod litery B (0x42)
z = ((char*)K)[2];              // z otrzymuje kod litery C (0x43)
p = (char*)K;                   // p wskazuje na początek tablicy K
x = sqrt((double)z);            // argument funkcji sqrt jest typu double
```

Pytania i zadania

3.4. Za pomocą operatora *sizeof* wyraź minimalny rozmiar struktury *S* zdefiniowanej jak w zad. 3.3. W argumencie operatora *sizeof* użyj:
a) nazwy struktury, b) nazwy zmiennej *S*, c) wskaźnika *pS*,
d) nazw typów.

3.5. Oblicz wartości i typy wyrażeń, jeśli zdefiniowano:
`float x=0.0, y=4.5;` `int k=-1, R[]={3,4,5,6};`

a) `!x+'A'`	b) `!(k+y)`	c) `R[!x+1]`
d) `!y`	e) `!!y`	f) `!R[0]`
g) `!k+y`	h) `!(!y+k)`	i) `R[3-!k]`

3.6. Oblicz wartości wyrażeń, a wyniki podaj w kodzie dziesiętnym lub szesnastkowym. Przyjmij, że zdefiniowano `int k=5;`.

a) `~0u`	b) `~!k`	c) `- ~k`
d) `~0ul-~0u`	e) `!~k`	f) `~ -k`
g) `~k`	h) `~(unsigned)k`	i) `~7u`

3.7. Oblicz wartości i typy wyrażeń jeśli zdefiniowano `float x=3.7, *y=&x, z=0;`

a) `-x`	b) `*(int*)y`	c) `!!z+!!x`
d) `-*y`	e) `~(unsigned)x`	f) `(char)x-*y`
g) `!*&z`	h) `~!*y`	i) `-++z`

3.8. Które z wyrażeń nie są poprawne jeśli zdefiniowano `float x, *y;`

a) `!++x`	b) `~*y`	c) `++*y`	d) `*--y`
e) `++(x+1)`	f) `~*(int)y`	g) `*y++`	h) `!y`
i) `~(x-1)`	j) `&y`	k) `*-y`	l) `(int)*y`

3.4. Operatory arytmetyczne

Operatory multiplikatywne:

mnożenie	***Wyr_a*1 * *Wyr_a*2**	
dzielenie	***Wyr_a*1 / *Wyr_a*2**	**(*Wyr_a*2 ### 0)**
reszta z dzielenia	***Wyr_c*1 % *Wyr_c*2**	**(*Wyr_c*2 ### 0)**

Operatory addytywne

dodawanie	***Wyr*1 + *Wyr*2**
odejmowanie	***Wyr*1 – *Wyr*2**

Argumenty mnożenia i dzielenia (*Wyr_a*1, *Wyr_a*2) muszą być typu arytmetycznego. **Dzielenie liczb całkowitych daje w wyniku liczbę całkowitą**.

Argumenty operatora % (*Wyr_c*1, *Wyr_c*2) muszą być typu całkowitego. Wynik reszty z dzielenia jest całkowity.

W dodawaniu i odejmowaniu oba argumenty (*Wyr*1, *Wyr*2) mogą być typu arytmetycznego, albo jeden z argumentów może być typu wskazującego i wtedy wynik jest tego samego typu wskazującego.

W odejmowaniu oba argumenty muszą być tego samego typu wskazującego. Różnica wyrażeń wskazujących jest typu całkowitego.

Przykłady

Wyrażenie	Wynik	Typ wyniku
8 * 3	24	`int`
8 * 3.0	24.0	`double`
8 / 5	1	`int`
8.0 / 5	1.6	`double`
14 % 3	2	`int`
15 +3.0	18.0	`double`

Jeśli zmienne *A* i *pA* zdefiniowano

```
float A[10], *pA=A;
```

Wyrażenie	Wynik	Typ wyniku
`A+5`	`&A[5]`	`float*`
`5+pA`	`&A[5]`	`float*`
`&A[8]-A`	`8`	`int`

Uwaga:

Mnożenie i dodawanie są działaniami przemiennymi i łącznymi. Kolejność opracowywania ich argumentów może być dowolnie zmieniana przez kompilator.

Przykłady

Wyrażenie	Możliwe wyniki	
`x++ * (x*y)`	`x*(x+1)*y`	`x*x*y`
`++x * (y*x)`	`(x+1)*y*(x+1)`	`(x+1)*y*x`
`++a + (b+a)`	`2a+b+2`	`2a+b+1`

Niezamierzone dzielenie całkowite w miejsce rzeczywistego (np. $x=k/10$; zamiast $x=k/10.0$;) może być przyczyną nieoczekiwanych (błędnych) wyników z programu.

Pytania i zadania

3.9. Oblicz wartość i typ wyrażenia

a) `3.5*2`	b) `7%4`	c) `7*5%4`	d) `9.0/4*5`
e) `5.0/2`	f) `7%4/2`	g) `7*5/4`	h) `7.0*5/4`
i) `5/2`	j) `7%4*5`	k) `9/4*5`	l) `7.0*(5/4)`

3.10. Oblicz wartość i typ wyrażenia jeśli zdefiniowano: `float x=15; int k=4;`

a) `k*y+10`	b) `k+0.5`	c) `&A[k+1]-A`
e) `y-k`	e) `(y+3)*(k+1)`	f) `&A[k]-&A[(int)y]`
g) `k+y`	h) `y*y-k*k`	i) `y/k-(int)y%k`

3.5. Przesunięcia bitowe

w lewo	***Wyr_c*1 << *Wyr_c*2**
w prawo	***Wyr_c*1 >> *Wyr_c*2**

Oba wyrażenia muszą być całkowite. Wyrażenie *Wyr_c*2 nie może być ujemne ani większe od liczby bitów wyrażenia *Wyr_c*1.

Typ wyniku jest zgodny z typem *Wyr_c*1 po konwersji standardowej. Wartością wyniku jest wyrażenie lewe *Wyr_c*1, którego bity przesunięto w lewo (<<) albo w prawo (>>) o *Wyr_c*2 bitów. Zwolnione bity są wypełniane zerami, ale jeśli *Wyr_c*1 jest ujemne, to przy przesunięciu w prawo zwolnione najstarsze bity nie są określone, ponieważ mogą być one zależnie od implementacji wypełniane bitem znaku albo zerami.

Przykłady

Przesunięcie w lewo

```
k =         11100011 11000011
k << 3 =    00011110 00011000
```

Przesunięcie w prawo

```
k =         11100011 11000011
k >> 3 =    00011100 01111000    gdy k jest typu unsigned
k >> 3 =    ???11100 01111000    gdy k jest typu int  (? = 0 lub 1)
```

Zauważmy, że przesunięcia bitowe liczb dodatnich o *N* mnożą (<<) albo dzielą (>>) te liczby przez *N*-tą potęgę liczby 2, o ile wynik mnożenia nie przekroczy zakresu wartości typu przesuwanej liczby. Przesunięcia wykonywane są dużo szybciej niż mnożenia lub dzielenia.

15 << 3 daje w wyniku tyle co 15×8 bo $8 = 2^3$,
15 >> 3 daje w wyniku tyle co 15/8.

Rotacja liczby ujemnej w prawo może być przyczyną złego działania programu.

Pytania i zadania

3.11. Oblicz wartości wyrażeń zakładając, że liczby typu *int* zapisane są na 16 bitach.

a) `1<<10` b) `7<<3<<1` c) `30<<3>>5`
d) `1+2<<3` e) `45000u>>5` f) `0xAF9785ul>>8`
g) `2<<3+1` h) `~1u>>12` i) `07235>>6`

3.12. Kiedy następujące pary wyrażeń są równoważne, a kiedy nie?

a) `k>>n<<m` `k<<m>>n` b) `k>>n` `k/(1<<n)` c) `k<<n` `k*(1<<n)`

3.13. Zinterpretuj sens wyrażeń:

```
a) k>>n<<n      b) 1u<<n      c) ~0u>>8*sizeof(int)-n
d) ~(~0u<<n)    e) ~(~k<<n)   f) (k<<2)+k
```

3.6. Operatory porównania i przyrównania

***Wyr*1 @ *Wyr*2**

gdzie **@** jest jednym z operatorów:

porównania: < <= > >= (mniejsze, nie większe, większe, nie mniejsze)
przyrównania: == != (równe, różne)

Argumenty *Wyr*1 i *Wyr*2 są oba arytmetyczne albo oba wskazujące. Wynik jest stałą typu *int* równą 0, gdy relacja jest fałszywa lub 1, gdy jest prawdziwa.

$$\text{Wyr1 @ Wyr2} = \begin{cases} 0 & \text{gdy relacja jest fałszywa,} \\ 1 & \text{gdy relacja jest prawdziwa.} \end{cases}$$

Przykłady

Wyrażenie	Wynik	Wyrażenie	Wynik
`5 >= 0`	`1`	`(0 < 3)-5`	`-4`
`3.1 < 1`	`0`	`1 != 5 > 0`	`0`
`0 == 0`	`1`	`(1 != 5) > 0`	`1`
`1 != 1`	`0`	`0 == 0 == 0`	`0`
`0 < 3-5`	`0`	`6 > 4 > 2`	`0`

Częstym błędem jest użycie operatora przypisania (=) w miejsce przyrównania (==).

Pytania i zadania

3.14. Oblicz wartości wyrażeń:

a) `1<2` b) `-2>0` c) `c) 8!=0` d) `7>0!=0`
e) `3<=3` f) `(3>2)+5` g) `g) 3>2+5` h) `1!=1<0.5`
i) `'A'==65` j) `1.7>=1.5` k) `k) -3<-2<0` l) `3>2==1<5`

3.15. Jakie wartości przyjmują następujące wyrażenia w zależności od *x* i *y*?

a) `(x>y)-(x<y)` b) `(x>-y)+(x==y)`
c) `(x>1)+(x<-1)` d) `(x>-1)+(x<1)-1`

3.7. Bitowe operatory logiczne

iloczyn	**(AND)**	***Wyr_c*1 & *Wyr_c*2**
różnica	**(EX-OR)**	***Wyr_c*1 ^ *Wyr_c*2**
suma	**(OR)**	***Wyr_c*1 | *Wyr_c*2**

Argumenty są wyrażeniami całkowitymi. Wynikiem jest stała całkowita. Wartość wyniku powstaje w efekcie wykonania operacji kolejno na parach pojedynczych bitów bez przeniesień. Jeśli *B*1 i *B*2 są parą bitów z tych samych pozycji w *Wyr_c*1 i *Wyr_c*2, to odpowiadający im bit wyniku jest następujący:

*B*1	*B*2	*B*1 & *B*2	*B*1 ^ *B*2	*B*1 | *B*2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

Bitowe operatory logiczne są łączne i przemienne, zatem wyrażenia mogą być przebudowywane podobnie jak w przypadku mnożenia i dodawania.

Przykłady

```
     j =  10000011 11000011 = 0x83c3
     k =  00000010 01110110 = 0x0276
j & k =  00000010 01000010 = 0x0242
j ^ k =  10000001 10110101 = 0x81b5
j | k =  10000011 11110111 = 0x83f7
```

Zauważmy, że

`k & 1 << n`	– testuje *n*-ty bit z *k*,
`k ^ 1 << n`	– daje w wyniku *k* z zanegowanym *n*-tym bitem,
`k & ~(1 << n)`	– daje w wyniku *k* z wyzerowanym *n*-tym bitem,
`k | 1 << n`	– daje w wyniku *k* z ustawionym *n*-tym bitem.

Stosowanie operacji bitowych (w tym też przesunięć) przyspiesza działanie programu i upraszcza kod wynikowy. Unikanie operacji bitowych kosztem nadużywania operacji arytmetycznych jest złym nawykiem.

Pytania i zadania

3.16. Oblicz wartość wyrażeń:

a) `26&28` b) `28&26<<1` c) `32|21&5`
d) `26^28` e) `28<<1^26` f) `2|31&~7^16`
g) `26|28` h) `28>>1|26` i) `65^27&14`

3.17. Uprość wyrażenia

a) `m &(k|~k)` b) `m|m` c) `m-m%8`
d) `m|k &~k` e) `m & m` f) `m^~0`
g) `m^m` h) `1<<n|1<<n+1` i) `~(~m/~k)`
j) `~(~m &~k)` k) `m &-k|k&~m` l) `m|m &k`

3.8. Operatory logiczne

koniunkcja (and)	***Wyr*1 && *Wyr*2**
alternatywa (or)	***Wyr*1 || *Wyr*2**

Argumentami są wyrażenia arytmetyczne lub wskaźniki. Wynikiem jest stała typu *int* o wartości 0 lub 1.

Wyrażenie prawe *Wyr*2 jest opracowywane tylko wtedy, gdy jest to konieczne, czyli wtedy, gdy lewy argument (*Wyr*1) w koniunkcji jest różny od zera, a w alternatywie jest równy zeru.

$$A \;\&\&\; B = \begin{cases} 0 & \text{gdy } A = 0 \text{ lub } B = 0, \\ 1 & \text{gdy } A \neq 0 \;\text{ i }\; B \neq 0, \end{cases}$$

$$A \;||\; B = \begin{cases} 0 & \text{gdy } A = 0 \;\text{ i }\; B = 0, \\ 1 & \text{gdy } A \neq 0 \text{ lub } B \neq 0. \end{cases}$$

Przykłady

Wyrażenie	Wynik
`-1 && 5`	`1`
`5 && 0`	`0`

Wyrażenie	Wynik
`5 || 0`	`1`
`0 || 0`	`0`

Uwaga: W poniższych wyrażeniach prawy argument nie musi być opracowywany.

```
a && b++
c || d++
if(fp && fscanf(fp,"%d",&k)) printf("K=%d",k);
```

Inkrementacja *b*++ będzie wykonana tylko, gdy *a*###0, natomiast *d*++ tylko, gdy *c*=0. Odczyt z pliku *fp* będzie realizowany tylko, gdy *fp* jest różne od *NULL*, czyli gdy plik jest poprawnie otwarty.

Pytania i zadania

3.18. Oblicz wyrażenia

a) `26&&28` b) `7<0||3>1` c) `!5||!0&&2+2`
d) `0||13.5` e) `5&&0||0||3` f) `!5<2&&!0==0`
g) `7<0&&3` h) `5&&!0||0` i) `!5==!2&&4==4`

3.19. Oblicz wyrażenia oraz *A* i *B* zakładając początkowe wartości *A*=*B*=0.

a) `++A&&B++` b) `A&&(A++||B++)` c) `A++||B++&&A++`
e) `A++&&B++` e) `(++A||B++)&&A++` f) `A++||++B&&++A`
g) `++A||++B` h) `++A||B++&&A++` i) `A++||A++||B++||B++`

3.20. Uprość wyrażenia

a) `A&&A&&A` b) `!A||A` c) `A<B||A==B`
d) `A||A||A` e) `A||B&&A` f) `!A||!B`
g) `!A&&A` h) `(A||B)&&A` i) `!A&&!B`

3.21. Zapisz w języku C warunki zakładając *double A, B, X, Y*; i *int K, J*;

a) $A \leq X \leq B$
b) $X < 0$ lub $X \geq 2$
c) *K* jest parzyste
d) $A \leq X \leq B$ lub $C \leq x < D$
e) $|X| < A$ i $X \geq 0$
f) *K* i *J* mają przeciwne znaki
g) *X* i *Y* mają jednakowe znaki
h) *K* nie jest podzielne przez 7
i) *K* podzielone przez 3 lub 10
j) *K* i *J* mają taką samą ostatnią cyfrę w zapisie heksagonalnym
k) *K* zawiera kod małej litery a *J* zawiera kod litery dużej
l) *K* i *J* zawierają kody tych samych liter

3.9. Operator warunkowy

***Wyr* ? *Wyr*1 : *Wyr*2**

Jest to operator trójargumentowy. W zależności od wartości wyrażenia *Wyr*, jeśli jest ona różna od zera ($Wyr \geq 0$), to opracowywane jest wyrażenie *Wyr*1, a w przeciwnym razie wyrażenie *Wyr*2. Wynikiem jest stała o wartości opracowanego wyrażenia (*Wyr*1 albo *Wyr*2). Typ wyniku nie zależy od tego, które wyrażenie zostało opracowane i jest równy szerszemu typowi *Wyr*1 lub *Wyr*2 po konwersji wstępnej. Wyrażenie *Wyr* może być arytmetyczne lub wskaźnikowe, natomiast *Wyr*1, *Wyr*2 mogą być oba równocześnie arytmetyczne albo oba wskaźnikowe.

Przykłady

`5 ? 3 : 0.7`	daje w wyniku 3.0 typu *double*,
`A>B ? A : B`	daje $max\{A, B\}$,
`A>B?&A:&B`	wskazuje na *A* albo na *B* zależnie od wyniku $A > B$,
`*(A>B?&A:&B)=0`	zeruje większą ze zmiennych *A* lub B.

Operator warunkowy ma niski priorytet i jest łączny prawostronnie, zatem wyrażenia

```
n>0 ? k+1 : 2.0+x*x
war1 ? wyr1 : war2 ? war3 ? wyr2 : wyr3 : wyr4
```

należy rozumieć jako

```
(n>0) ? (k+1) : (2.0+x*x)
war1 ? wyr1 : (war2 ? (war3 ? wyr2 : wyr3) : wyr4)
```

Pytania i zadania

3.22. Oblicz wartości i typy wyrażeń

a) `7.5?'A':'a'` b) `0.1?3L:1.5` c) `1-2?5/2.5:3>0+2`

d) `1+2?2*3:7.5` e) `1?4?10:9:8` f) `1?0?2:3?4:5.5:'A'`

3.23. Oblicz wartości wyrażeń oraz A i B, zakładając początkowe wartości $A=B=0$

a) `A==0?++A:B++`

b) `A!=B?++A+1:B+++5`

c) `A||B?A+++B:A<=B?B++:A+20`

3.10. Operatory przypisania

LW* = *Wyr
LW* @= *Wyr // ***LW* = (*LW*) @ (*Wyr*)**

gdzie @ jest jednym z operatorów: ***** **/** **%** **+** **–** **<<** **>>** **&** **^** **|**

LW jest *l*-wartością (np. nazwą zmiennej),
Wyr jest wyrażeniem, którego typ ma standardową konwersję do typu *LW*

Wynikiem jest stała o wartości wyrażenia (*LW*) @ (*Wyr*) konwertowanej do typu *l*-wartości *LW*. Argument *LW* otrzymuje wartość wyniku.

Przykłady

Dla każdego przykładowego wyrażenia przyjmijmy następujące definicje

```
int j, k=2;          float y, x=3.5;
```

Wyrażenie	Wynik	Typ wyniku	Efekty uboczne	Uwagi
`y=x`	`3.5`	`float`	`y=3.5`	
`i=k=y=x`	`3`	`int`	`y=3.5, k=3, i=3`	
`y=j=x`	`3.0`	`float`	`j=3, y=3`	`y=j`
`x*=k+1`	`10.5`	`float`	`x=10.5`	`3.5*3`
`k+=x`	`5`	`int`	`k=5`	`k=2+3.5`
`k <<= (j=x)`	`16`	`int`	`j=3, k=16`	`k=2<<3`
`x*=x`	`12.25`	`float`	`x=12.25`	`x=3.5*3.5`
`x*=k+(j=y=x+1)`	`21`	`float`	`y=4.5, j=4, x=21`	`3.5*6`
`(j=x) > 0`	`1`	`int`	`j=3`	`3>0`
`(y=x) == (j=x)`	`0`	`int`	`y=3.5, j=3`	`3.5==3`

Przykłady wykorzystania wyników przypisania w instrukcjach *if*:

```
if((c=getch())==27) break;
if((fp=fopen("dane.txt","r"))==NULL)
                        printf("Brak pliku dane.txt");
if((buf=malloc(dl))==NULL)
                        printf("Brak pamieci lub dl<=0");
```

W powyższych przykładach wyniki funkcji *getch*, *fopen*, oraz *malloc* są podstawiane kolejno pod zmienne *c*,*fp* oraz *buf*. a następnie wyniki tych podstawień są przyrównywane do kodu klawisza *Esc* równego 27 lub do wskaźnika pustego NULL.

Przykłady zmiany bitów w słowie

`k ^= 1 << n;`	neguje *n*-ty bit w *k*,
`k &= ~(1 << n);`	zeruje *n*-ty bit w *k*,
`k |= 1 << n;`	ustawia *n*-ty bit w *k*.

Wielokrotne podstawienia (jak np. $j=k=x=y=0$**) są dozwolone, ponieważ operator przypisania daje wynik, podobnie jak dają go inne operatory (**np.: + – * /**).**

Używanie konstrukcji $x=x@y$ zamiast $x@=y$ utrudnia optymalizację kodu programu. Budowanie skomplikowanych wyrażeń i nadużywanie efektów ubocznych zmniejsza przejrzystość programu i utrudnia jego uruchamianie.

Pytania i zadania

3.24. Rozpisz każde wyrażenie na ciąg wyrażeń prostych (z jednym operatorem przypisania)

a) `x=y+=10` b) `x*=j<<=3` c) `j=(c=getch())==27`
d) `k=(x+=5)>(y/=3)` e) `x=y+=a*=b=c` f) `k=x+=5>(y|=3)`

3.25. Oblicz wartości i typy wyrażeń oraz wartości zmiennych, jeśli zdefiniowano
`int k=4, n;  float x=2.5, y, Z[8]={1, 2};  char C;`
Rozpisz każde wyrażenie na ciąg wyrażeń prostych (z jednym operatorem przypisania).

a) `y=x*=2` b) `C=k+'A'` c) `n=k>x`
d) `n=x+=7` e) `k*=x++` f) `(n=k)>x`
g) `y=n=k/=x-=1` h) `Z[k++]=Z[0]+=3` h) `Z[n=x]=Z[k-=3]+=5`

3.11. Operator połączenia

***Wyrażenie*1, *Wyrażenie*2**

Opracowywane są kolejno wyrażenia od lewej do prawej. Typ i wartość wyniku wyrażenia połączenia są równe typowi i wartości wyrażenia prawego, obliczanego jako ostatnie.

Przykłady

```
k = (-5, 6, 8)          // podstawia k=8
k = (i=0, j=1, n=2)     // podstawia i=0, j=1, n=2, k=2
A[4, 6]                 // jest równoważne A[6]
```

Operator połączenia jest najczęściej stosowany w instrukcji pętli *for*, np.

```
for (x=0.0, k=1, j=n; k<n; k++, j--)
```

Przed rozpoczęciem pierwszej iteracji wykonywane są trzy przypisania: $x=0$, $k=1$, $j=n$. Po każdej iteracji wykowywane są dwie inkrementacje: k++, j-- .

Inicjując odpowiednio zmienne globalne można uruchamiać różne funkcje przed uruchomieniem funkcji *main*. Na przykład, jak pokazano poniżej, można wyczyścić ekran podczas inicjowania zmiennej k.

```
int k = (clrscr(), 6);          // czyści ekran i podstawia k=6
```

Pytania i zadania

3.26. Wyrażenia z zadań 3.23 i 3.24 zapisz w postaci wyrażenia przypisania, w którym prawnym argumentem będzie wyrażenie połączenia.

3.27. Wyjaśnij różnicę między wyrażeniami:

a) `A[i,j]` `A[i][j]` b) `B[k]+=a` `B[k]=B[k]+a` c) `B[k++]+=a` `B[k++]=B[k++]+a`

3.28. Podaj kolejność opracowywania i typy wyrażeń. Przyjmij następujące typy zmiennych:

```
int j, k[20];  float x, y, z[20];  char p,*q, r[80];
long L;
```

a) `-x/n*2`
b) `*y++`
c) `++*y`
d) `j|n&*k^7`
e) `x+=j>>=n<=1`
f) `*k<<2`
g) `x<1==n>=1`
h) `j!=z[1]>*y`
i) `r[sizeof(float)+1]++`
j) `!(int)x?&j:k+n`
k) `j%n>>1<=*y++`
l) `*q=k[3]%=1+!!x<<7`

3.29. Wymień operatory 2-argumentowe, których argumenty:
a) oba muszą być typu całkowitego,
b) oba mogą być typu wskazującego,
c) jeden z nich musi być typu wskazującego,
d) jeden musi być *l*-wartością.

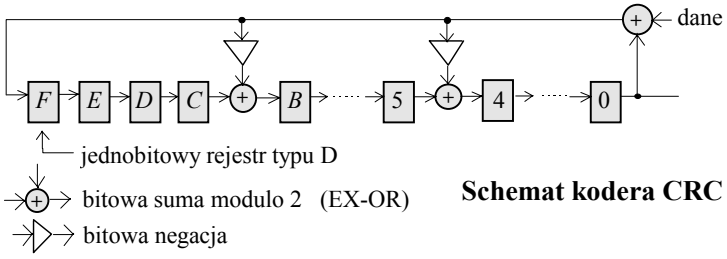
Zadania laboratoryjne

3.30. Sprawdź praktycznie:
a) czy operator indeksowania [ ] jest przemienny?
b) czy kolejność opracowywania składników sumy $a+b+c+d$ zależy od umieszczenia w niej nawiasów?

3.31. Sprawdź w twojej implementacji języka C:
a) jak wypełniane są zwalniane pozycje podczas przesuwania liczby ujemnej w prawo >>,
b) w jakiej kolejności opracowywane są argumenty wywołania funkcji?

3.32 Napisz program, który dla podanych dwu liczb całkowitych wypisze numery tych pozycji binarnych, na których różnią się te liczby.

3.33. Napisz funkcję, która dla danej tablicy znakowej obliczy 16-bitową sumę kontrolną CRC wg wielomianu $x^{16} + x^{12} + x^5 + 1$. Podany na schemacie koder zrealizować w jednej zmiennej całkowitej



Schemat kodera CRC

Po wprowadzeniu ostatniego bitu danych (od najmłodszego do najstarszego) rejestr przesuwny *FED*...210 zawiera 16-bitową sumę kontrolną CRC.

4. Instrukcje

4.1. Instrukcje wyrażeniowe

4.1.1. Instrukcja wyrażeniowa

***Wyrażenie*;**

Instrukcja wyrażeniowa to wyrażenie zakończone średnikiem. Średnik jest tu ogranicznikiem instrukcji, nie zaś separatorem jak w języku Pascal.

Wykonanie instrukcji wyrażeniowej to opracowanie wyrażenia i odrzucenie jego wyniku. Na przykład wykonanie instrukcji $k=2;$ polega na przypisaniu zmiennej *k* wartości 2. Wynikiem jest tu stała typu takiego jak typ zmiennej *k*. Zmienna *k* przyjmuje wartość wyniku, natomiast sam wynik nie jest nigdzie wykorzystany.

Przykłady

```
k=j=0;
k++;
printf("K=%d", k);
```

4.1.2. Instrukcja pusta

```
;
```

Instrukcję pustą tworzy średnik, przed którym nie stoi żadne wyrażenie.

Przykłady

```
for(i=0; A[i]=B[i]; i++) ;
while(*p++ = *q++) ;
for(i=0; i<n; i++); scanf(”%lf”, &A[i]);
```

W powyższych instrukcjach kopiowania tekstów wewnętrzne instrukcje pętli są puste – końcowe średniki nie są poprzedzone wyrażeniami. Funkcja *scanf* zostanie wywołana jeden raz dla $i=n$, ponieważ wewnętrzną instrukcją pętli *for* jest tu instrukcja pusta utworzona przez średnik po nawiasie okrągłym zamykającym.

4.1.3. Instrukcja grupująca

{ ***Instrukcja***
Instrukcja
. . . . . .
Instrukcja
}

Instrukcja grupująca jest ciągiem instrukcji, zamkniętym w nawiasy klamrowe { }. Instrukcja grupująca jest instrukcją – traktuje się ją składniowo jak pojedynczą instrukcję wyrażeniową.

Jeśli w instrukcji grupującej wystąpią definicje lub deklaracje, to taka instrukcja nazywa się blokiem.

4.1.4. Etykiety

Etykieta jest nazwą i jest od instrukcji oddzielona dwukropkiem. Instrukcja z etykietą jest instrukcją i może być poprzedzona kolejną etykietą. Etykiet się nie definiuje.
Nazwy etykiet nie kolidują z nazwami zmiennych ani funkcji i nie przesłaniają tych nazw.

Etykieta* : *Instrukcja

Przykłady

```
E1:X+=5;
E2:{ int i;
       for(i=0; i<N; i++) X+=A[i];
   }
E3:E4:printf("%lf", X);
K:K=5;
```

W przykładzie etykietami są: *E*1, *E*2, *E*3, *E*4 oraz *K*. Występuje też zmienna *K*.

Pytania i zadania

4.1. Wypisz instrukcje wyrażeniowe, puste i grupujące z przytoczonego fragmentu programu

```
{ int i, n=10;
  double x, y;
et1:i=0;;
  while(i<n)
    { y=i*i+1;
      x+=y*y;
      i++;
    };
et2:et3:printf("%lf", x);
}
```

4.2. Instrukcja if–else

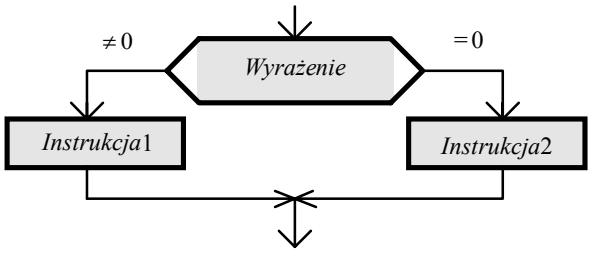
if (*Wyrażenie***)** *Instrukcja*1
if (*Wyrażenie***)** *Instrukcja*1 **else** *Instrukcja*2

Jeśli *Wyrażenie* ma wartość różną od zera, to wykonywana jest *Instrukcja*1, w przeciwnym razie wykonywana jest *Instrukcja*2, jeśli ona istnieje.

Schemat blokowy instrukcji *if–else* przedstawiono na rys. 4.1.

Przykłady

```
if(i>0) K[i]=1;
if(A>B) A=B; else B=A;
```



Rys. 4.1. Schemat blokowy instrukcji **if–else**

Gdy *Instrukcja*1 lub *Instrukcja*2 w instrukcji **if** są też instrukcjami **if**, to słowo **else** i jego *Instrukcja*2 odnoszą się zawsze do najbliższego poprzedzającego **if**, które nie jest skojarzone z żadnym **else**. Jeśli brakuje słowa **else**, to przyjmuje się, że *Instrukcja*2 jest instrukcją pustą.

Przykłady

Instrukcja **if**	Interpretacja
`if(i>0) if(a>b) K[i]=a;` `  else b=a;`	`if(i>0)` `  {if(a>b) K[i]=a; else b=a;}`

`if(i>0) if(a>b)`
`  K[i]=a; else b=a;`
`  else a=b;`
	`if(i>0)`
	`  {if(a>b) K[i]=a; else b=a;}`
	`  else a=b;`

`if(i>0) if(a>b) K[i]=a;`
`  else`
`  if(a>0) b=a; else a=b;`
	`if(i>0)`
	`  {if(a>b) K[i]=a; else`
	`  {if(a>0) b=a; else a=b;}`
	`  }`

`if(a>b && a>C) ; else a=0;`	`if(!(a>b && a>c)) a=0;`

Pytania i zadania

4.2. Połącz słowa **else** z odpowiednimi słowami **if**

a) `if(d>=0) if(d>0) x=e+sqrt(d); else x=e;`

b) `if(d<0) x=0; else if(d>0) x=e+sqrt(d); else x=e;`

c) `if(k>0) if(k<M) k++; else k=0; else k=M;`

d) `if(k>0) if(k<M) k++; else if(k==M); else k=0; else`
`if(k) ; else k=M;`

4.3. Narysuj schematy blokowe instrukcji z zadania 4.2.

4.4. Instrukcje a) i d) z zadania 4.2 zapisz prościej, usuwając instrukcje puste i łącząc warunki.

4.5. Usuń zbędne nawiasy klamrowe i narysuj schematy blokowe instrukcji

a) `if(k) {if(d>0.0) x=d;} else {d=0.0; k--;}`

b) `if(k>0){if(d>0.0) d=-d; else d=0.0;} else{if(k<0) k=0;}`

c) `if(k&&n){k+=n; n++;}else{if(k) n++; else{x=0.0; k++;}}`

4.3. Instrukcja switch

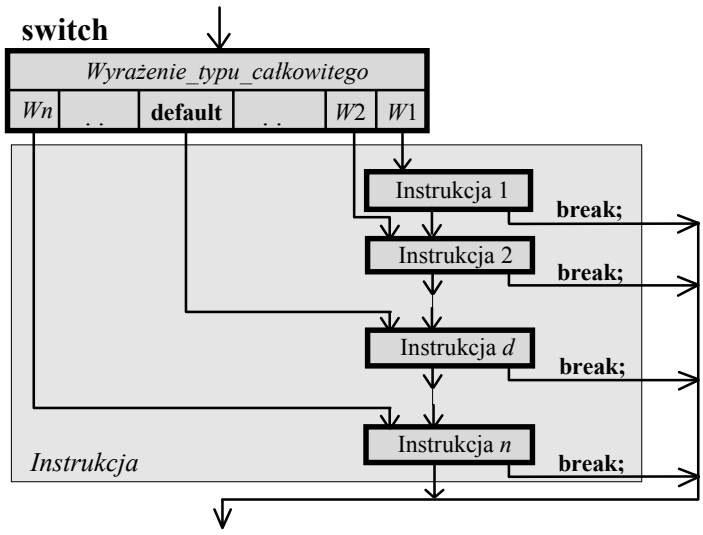
switch (*Wyrażenie_typu_całkowitego***)** *Instrukcja*

Instrukcja jest zwykle instrukcją grupującą, której instrukcje wewnętrzne mogą być poprzedzone etykietami postaci

case *Wyrażenie_stałe***:**
default:

Tylko jedna z etykiet może mieć postać *default:*.

Wartość *Wyrażenia_typu_całkowitego* w nawiasach instrukcji **switch** porównuje się z *Wyrażeniami_stałymi* etykiet **case**. Jeśli zostanie stwierdzona równość, następuje skok do instrukcji poprzedzonej tą etykietą. W przeciwnym razie następuje skok do instrukcji poprzedzonej etykietą **default** o ile taka istnieje, lub skok na koniec instrukcji **switch**, jeśli brak jest etykiety **default**. Wewnątrz instrukcji **switch** można używać instrukcji **break;**. Powoduje ona skok na koniec instrukcji **switch**, jak pokazano na rys. 4.2.



Rys. 4.2. Schemat blokowy instrukcji **switch**

Przykład 4.1

Obliczyć

$$y = \begin{cases} x^2 & \text{dla } n = 2 \\ x & \text{dla } n = 1 \\ \sin x & \text{dla } n = 3 \\ \cos x & \text{dla } n = 4 \\ 1 & \text{dla innych } n \end{cases}$$

Rozwiązanie:

```
y=1;
switch(n)
  { case 3:y=sin(x);
           break;
    case 4:y=cos(x);
           break;
    case 2:y*=x;
    case 1:y*=x;
           break;
  }
```

Ostatnia instrukcja *break*; nie jest potrzebna. Umieszczono ją, ponieważ może być ona konieczna, gdy zajdzie potrzeba dopisania dalszych przypadków *case*, aby nie ingerować w przypadki już przetestowane i uruchomione.

Przykład 4.2

Wykonać:	$x=y;\ n=0;$	gdy k jest podzielne przez 8,
	$x=y;$	gdy k jest podzielne przez 4,
	$y=x;\ n=1$	gdy k jest parzyste, ale nie jest podzielne przez 4,
	$y=0$	w pozostałych przypadkach.

Rozwiązanie:

```
switch(k&7)
  { case 0:n=0;
    case 4:x=y;
           break;
    case 2:
    case 6:y=x;
           n=1;
           break;
    default:y=0;
           break;
  }
```

Wyrażenie *k*&7 daje resztę z dzielenia *k* przez 8 (tak jak *k*%8). Podobnie jak w poprzednim przykładzie ostatnią instrukcję *break*; umieszczono zwyczajowo.

Częstym błędem jest brak instrukcji **break;** pomiędzy przypadkami instrukcji *switch*.

Pytania i zadania

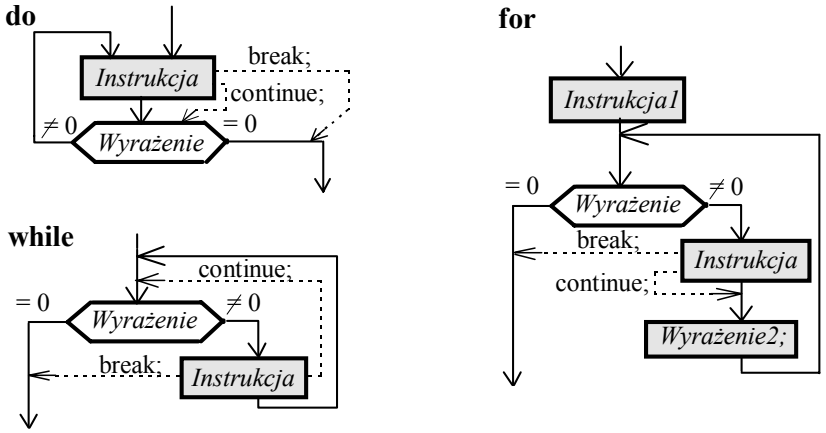
4.6. Narysuj schemat blokowy instrukcji *switch* z przykładu 1.

4.7. Wielomian $y=a_3 x^3+a_2 x^2+a_1 x+a_0$ można obliczyć ze wzoru $y=((a_3 x+a_2)x+a_1)x+a_0$. Użyj instrukcji *switch* do obliczania wartości wielomianów stopnia od 0 do 4.

4.8. Napisz instrukcję *switch*, która zmiennej *K* nada wartość 2, gdy naciśnięto klawisz *'t'* lub *'T'*, wartość 1, gdy naciśnięto klawisz *'n'* lub *'N'*, -1 gdy *Esc*, 3 gdy *Enter* oraz wartość 0, gdy naciśnięto inny klawisz. Kod klawisza zwraca funkcja *getch*( ).

4.4. Pętle do, while i for

do *Instrukcja* **while (***Wyrażenie***)**;
while (*Wyrażenie***)** *Instrukcja*
for (*Wyrażenie*1; *Wyrażenie*; *Wyrażenie*2**)** *Instrukcja*



Rys. 4.3. Schematy blokowe pętli **do**, **while** i **for**

We wszystkich rodzajach pętli *Instrukcja* jest wykonywana tak długo, jak długo *Wyrażenie* jest różne od zera. Ten warunek jest sprawdzany w pętli **do** po wykonaniu instrukcji, natomiast w pętlach **while** i **for** przed wykonaniem instrukcji (rys. 4.3). Pętla **for** jest to pętla **while** uzupełniona o instrukcję *Instrukcja*1 wykonywaną przed wejściem do pętli oraz o instrukcję *Wyrażenie*2; wykonywaną po każdej iteracji (każdorazowym wykonaniu instrukcji *Instrukcja*). Jeżeli wyrażenia *Wyrażenie* są puste, to są interpretowane jako różne od zera, czyli prawdzie.

Instrukcja **break;** powoduje przerwanie wykonywania pętli.

Instrukcja **continue;** powoduje przerwanie bieżącej iteracji, czyli skok na koniec zapętlonej instrukcji *Instrukcja*, która jest wtedy instrukcją grupującą. W pętli **for** jest to skok do instrukcji *Wyrażenie*2;.

Przykład 4.3

Wczytywanie liczby x z przedziału od *Min* do *Max*. Po wczytaniu liczby jest ona sprawdzana. Jeśli $x < Min$ lub $x > Max$, to czytanie powtarza się.

```
do
  { printf("\nPodaj x o wartosci od %lf do %lf: ", Min, Max);
    scanf("%lf", &x);
  } while(x<Min || x>Max);
```

Przykład 4.4

Oczekiwanie na naciśnięcie klawisza T lub N.

```
printf(" (T/N): ");
do {z=toupper(getch());} while(z != 'T' && z != 'N');
```

Obliczanie sumy $S = A_0 + A_1 + ... + A_n$

```
for(S=i=0; i<n; i++) S+=A[i];
```

Przykład 4.5

Obliczanie sumy $S = 1 + x + x^2 + x^3 + ... + x^n + ...$ aż do momentu, gdy $x^n \le Eps$.

```
S=1.0;
y=x;
```

```
while(y>Eps) {S+=y; y*=x;}
```

lub

```
for(S=1.0, y=x; y>Eps; y*=x) S+=y;
```

Przykład 4.6

Obliczanie wartości średniej liczb dodatnich i wartości średniokwadratowej liczb wszystkich do pierwszej liczby zerowej.
Z wykorzystaniem instrukcji *break*; i *continue*;

```
for(s1=s2=0.0,i=j=0;i<n;i++)
 { if(x[i]==0.0) break;
  s2+=x[i]*x[i];
  if(x[i]<0.0) continue;
  s1+=x[i];
  j++;
 }
if(i) s2/=i;
if(j) s1/=j;
```

Bez instrukcji *break*; i *continue*;

```
for(s1=s2=0.0,j=i=0;i<n&&x[i]!=0;i++)
 { s2+=x[i]*x[i];
  if(x[i]>0.0)
    { s1+=x[i];
      j++;
    }
 }
 if(i) s2/=i;
 if(j) s2/=j;
```

Przykład 4.7

Funkcje *fun*1, *fun*2 oraz *fun*3 należy wykonywać w pętli w odpowiedzi na naciskanie odpowiednio klawiszy 1, 2 oraz 3 aż do naciśnięcia klawisza *Esc*, który powoduje wyjście z pętli. Pozostałe klawisze należy ignorować. Ponieważ dla ogólności rozważań nie podano konkretnych funkcji, argumenty funkcji zastąpiono trzema kropkami.

```
do
 { clrscr();
   cprintf("Naciśnij klawisz z numerem funkcji lub Esc ");
   switch(z=getch())
     { case '1':     fun1( ... );
                     break;
       case '2':     fun2( ... );
                     break;
       case '3':     fun3( ... );
                     break;
     }
 } while(z!=27);
```

Bardzo częstym błędem jest wstawianie zbędnego średnika, a więc instrukcji pustej jako instrukcji pętli *for*.

Pytania i zadania

4.9. Zapisz w postaci instrukcji pętli

a) $y = \sum_{i=0}^{n} (a_i - b)^2$ b) $y = \sum_{i=0}^{n} a_i x^i$ c) $y = \sum_{j=0}^{m} \sum_{i=0}^{n} a_{ij}$

4.10. Oblicz sumy, biorąc pod uwagę tylko składniki o wartościach bezwzględnych większych niż epsilon

a) $y = \sum_{n=1}^{\infty} \frac{1}{n^2}$ b) $y = y + \sum_{n=1}^{\infty} \frac{x}{n^2}$ c) $y = \sum_{n=0}^{\infty} \frac{x^n}{n!} (-1)^2$

4.11. Napisz pętlę, która będzie czytać (i ignorować) naciskane klawisze aż do chwili, gdy zostanie naciśnięty klawisz *T*, *N* albo *Esc*.

4.12. Napisz pętlę, która będzie wyprowadzać kody naciskanych klawiszy aż do naciśnięcia klawisza *Esc* o kodzie 27. Kod klawisza *Esc* nie powinien być wyprowadzony.

4.13. Napisz pętlę, która policzy ile liter *a* lub *A* jest w danym tekście.

4.14. Sprawdź, czy we wskazanym tekście jest litera *k* lub *K*. Użyj instrukcji *break*;.

4.15. Funkcja *kbhit*( ) daje w wyniku liczbę różną od zera, gdy w buforze klawiatury znajduje się nie odczytany znak. Funkcja *getch*( ) pobiera ten znak z bufora. Napisz pętlę, która opróżni bufor klawiatury ze znaków.

4.5. Instrukcja skoku

goto *Etykieta*;

Instrukcja skoku powoduje przeniesienie sterowania do instrukcji poprzedzonej określoną etykietą.

Do dobrego stylu programowania należy unikanie instrukcji skoku, ponieważ najczęściej zaciemnia ona przejrzystość algorytmu. Konstrukcja języka C, podobnie jak wielu innych języków, umożliwia pisanie programów bez użycia instrukcji **goto**.

W pewnych sytuacjach użycie instrukcji **goto** jest zalecane. Do takich należy obsługa błędów. Instrukcja **goto** pozwala w sposób elegancki przerwać wykonywanie obliczeń w dowolnym miejscu. Należy jednak skakać tylko do przodu. Skoki wstecz naruszają przejrzystość i strukturalność programu.

Przykład 4.8

Niech pewna funkcja otwiera kilka plików i alokuje pamięć na tablicę liczb. Przed wyjściem z funkcji należy te pliki pozamykać, a przydzieloną pamięć zwolnić. W przypadku błędu lub niepowodzenia przed wyjściem z funkcji należy zamknąć tylko te pliki, które zostały otwarte i zwolnić tylko tę pamięć, która została zaalokowana.

```
  int i, j, n, *k;              // definicja wymiarów tablicy i jej wskaźnika
  FILE *fp1, *fp2;                    // definicja wskaźników do plików
  fp1=fopen("DANE","r");              // otwarcie pliku z danymi do odczytu
  if(fp1==NULL) goto E1;              // skocz, jeśli pliku nie otwarto
  j=fscanf("%d", &n);                                   // wczytanie n
  if(j<1) goto E2;                    // skocz, jeśli nie odczytano n
  k=calloc(n, sizeof(k[0]));          // pobierz pamięć na n liczb typu int
  if(k==NULL) goto E2;                // skocz, gdy brakuje pamięci
  for(i=0; i<n; i++)
    if(fscanf("%d",k+i)<1) goto E3;           // skocz, gdy błąd odczytu
  . . . . . .                                 // przetwarzanie danych
  fp2=fopen("WYNIKI","w");                    // otwarcie pliku na wyniki
  if(fp2==NULL) goto E3;              // skocz, gdy pliku nie otwarto
  for(i=0;i<n;i++)
    if(fprintf("%d ",k[i])<1) goto E4;        // skocz, gdy błąd zapisu
  . . . . . .
E4:fclose(fp2);                       // zamknięcie pliku z wynikami
```

```
E3:free(k);                    // zwolnienie pamięci
E2:fclose(fp1);                // zamknięcie pliku z danymi
E1:return;                     // wyjście z funkcji
```

Rezygnacja z instrukcji skoku pociągnęłaby rozbudowę programu o funkcję obsługi błędnych sytuacji albo o powielanie wywołań funkcji *fclose* i *free* oraz instrukcji *return* po każdym badaniu poprawności wykonania operacji. Użycie instrukcji skoku nie pogarsza w tym przypadku czytelności programu. Przerwanie realizacji algorytmu (przeskok części algorytmu) jest naturalną konsekwencją wystąpienia błędu.

Zaleca się tylko skoki do przodu. Zauważ, że kolejność czterech ostatnich zaetykietowanych instrukcji nie jest przypadkowa oraz że te instrukcje wystąpią, nawet gdy nie będzie się wykrywać ani obsługiwać błędów.

Pytania i zadania

4.16. Narysuj schemat blokowy funkcji z przykładu 4.8 z obsługą błędów.

Zadania laboratoryjne

4.17. Napisz program, który podane liczby od 1 do miliarda będzie wypisywać słownie. Uwzględnij zasady gramatyczne języka polskiego.

4.18. Napisz program, który z zadaną dokładnością obliczy pierwiastek kwadratowy p z liczby y posługując się wzorem iteracyjnym: $p_{n+1}=(p_n+y/p_n)/2$

4.19. Dla tablicy A o N wierszach i M kolumnach oblicz sumy elementów w każdym wierszu.

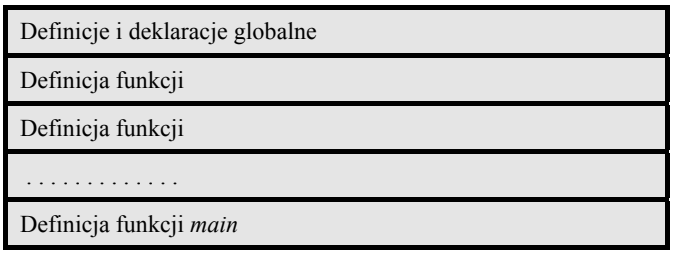
4.20. W N-elementowej tablicy są liczby całkowite uporządkowane rosnąco. Oblicz sumę (S) liczb, średnią arytmetyczną (m_1) liczb podzielnych przez 3 i średnią arytmetyczną (m_2) liczb podzielnych przez 6. W obliczeniach uwzględnij tylko liczby mniejsze lub równe K. Użyj instrukcji *break*; i *continue*;.

4.21. Napisz program, który będzie rozwiązywać trójkąty podanymi metodami *bbb*, *bkb* lub *kbk*. Zapętl program i na początku wstaw menu proponujące do wyboru jedną z powyższych metod albo zakończenie obliczeń. Zabezpiecz program przed przyjęciem nieprawidłowych danych.

4.22. Napisz program, który będzie wczytywać liczby dodatnie i wpisywać je do tablicy liczb rzeczywistych (*float* albo *double*) tak, aby były one w tej tablicy uporządkowane rosnąco. Program nie powinien przyjmować liczb ujemnych. Powinien zakończyć wprowadzanie po podaniu liczby zero lub po podaniu pewnej ustalonej (maksymalnej) ilości liczb. Po zakończeniu wprowadzania program powinien wypisać te liczby na ekranie.

5. Funkcje

Program w języku C jest zbudowany z funkcji definiowanych globalnie na jednym poziomie. Nie można definiować lokalnej funkcji wewnątrz innej funkcji. Różne funkcje mogą być pisane w oddzielnych plikach i mogą być niezależnie kompilowane. Każdy program musi zawierać główną funkcję o nazwie *main*, od której zaczyna się sterowanie obliczeniami (rys. 5.1).



Rys. 5.1. Struktura programu

5.1. Budowa funkcji

Funkcja składa się z nagłówka oraz instrukcji grupującej. W nagłówku definiuje się typ wyniku i nazwę funkcji oraz w nawiasach okrągłych listę parametrów. Każdy parametr ma określony typ. Dla każdego parametru jest automatycznie definiowana lokalna zmienna.

Zmienne mogą być definiowane na zewnątrz wszystkich funkcji i takie zmienne nazywa się *globalnymi*. Istnieją one przez cały czas wykonywania programu i mogą być dostępne wewnątrz każdej funkcji, która nie definiuje lokalnych zmiennych o takich samych nazwach co nazwy zmiennych globalnych. Zasięg zmiennych globalnych rozciąga się do końca pliku, w którym zostały one zdefiniowane lub zadeklarowane ze specyfikacją *extern*.

Zmienne definiowane wewnątrz funkcji są zmiennymi *lokalnymi*, dostępnymi tylko w bloku, w którym zostały zdefiniowane. Jeśli nazwa zmiennej lokalnej pokrywa się z nazwą zmiennej globalnej, to zmienna globalna jest przysłonięta i niewidoczna w tym bloku, w którym zdefiniowana jest ta zmienna lokalna.

Automatyczne zmienne lokalne powstają w momencie wejścia do bloku, w którym są one zdefiniowane, a ich wartości nie są określone. Z chwilą wyjścia z tego bloku, zmienne automatyczne przestają istnieć. Inicjowane zmienne automatyczne otrzymują swoje wartości początkowe za każdym razem, gdy powstają.
Zmienne statyczne lub **zewnętrzne** istnieją przez cały czas pracy programu. Swoje wartości początkowe otrzymują tylko raz w momencie uruchomienia programu.

Przykład 5.1

Poniższy program wyprowadzi napis 113, 160.

```
#include <stdio.h>
int k=50, j=10;
int funkcja(void)
  {return(k+100);}      // obszar zasięgu zmiennej globalnej k=50
void main(void)
  {int k=3;
   printf("\n%d, %d", k+100+j, funkcja()+j);
  }
```

Pytania i zadania

5.1. W podanym poniżej programie:
   a) Wypisz zmienne globalne i lokalne podając ich nazwy połączone z inicjatorami. Zaznacz dla każdej zmiennej jej zasięg.
   b) Zaznacz strukturę programu – określ zasięg definicji globalnych, zasięg definicji funkcji *fun i* zasięg definicji funkcji *main* (rys 5.1).
   c) Jaki napis zostanie wyprowadzony przez program.

```
#include <stdio.h>
int i=10, j=200, k=300;

void fun(void)
  {  int i=4;    static int j=0;   i++;  j++;
     printf("\nI=%d   J=%d   K=%d", i, j, k);}

void main(void)
  {int i, k=0;   for(i=0; i<3; i++) f(); }
```

5.2. Argumenty funkcji

Wywołanie funkcji ma postać

***nazwa funkcji* (*argument, argument, ..., argument*);**

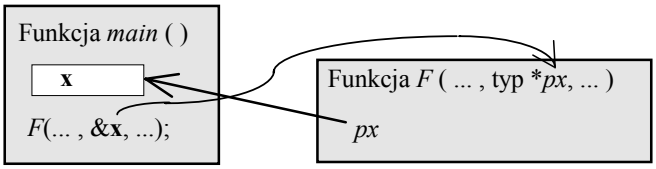
Liczba argumentów wywołania funkcji musi być zgodna z liczbą jej parametrów. Pomiędzy typami argumentów a typami odpowiadających im parametrów muszą istnieć konwersje standardowe.
Argumenty są przekazywane przez wartość. (W języku C++ jest możliwość przekazywania przez referencję czyli przez zmienną.)
Jeśli parametr ma posłużyć do wyprowadzenia wyniku określonego typu, to ten parametr musi być wskaźnikiem tego typu (lub w języku C++ referencją).

Na przykład argument może być typu *char* a parametr typu *int* i odwrotnie, ale argument i parametr nie mogą być: jeden np. typu *int* a drugi typu *int** (wskaźnik).

Przekazując wskazanie (programowy adres) zmiennej do funkcji dajemy jej możliwość zapisu pod to wskazanie, a co za tym idzie, możliwość zapisu do tej zmiennej. Z tego powodu argumentami np. funkcji *scanf* są wskaźniki do zmiennych, nie zaś same zmienne.

Jeśli np. przekażemy funkcji wartość wskaźnika *px*, to wewnątrz funkcji można odwoływać się do zmiennej *x*, tak jak pokazano na rysunku rys. 5.2. Wskaźnik *px* wewnątrz funkcji wskazuje na zmienną *x* zdefiniowaną poza funkcją, tak więc wynikiem wyrażenia **px* jest właśnie ta zmienna *x*. Wszędzie tam, gdzie wewnątrz funkcji trzeba użyć zmiennej zewnętrznej *x*, należy użyć wyrażenia **px*.

Gdyby *px* wskazywało na element *X*[0] tablicy zewnętrznej *X*, to element *X*[*i*] byłby wskazywany przez wyrażenie *px*+*i* (tożsame z &*px*[*i*]), zatem byłby wewnątrz funkcji osiągalny jako *(*px*+*i*) czyli jako *px*[*i*].



Rys. 5.2. Przekazanie wyniku przez wskaźnik

Przykład 5.2

Poniższy program wyprowadzi napis $i=7 \quad j=3 \quad k=16$.

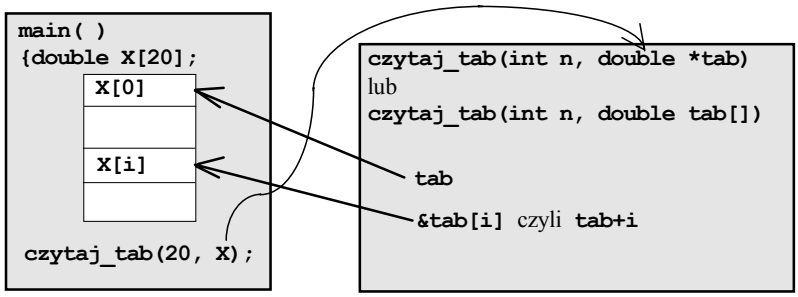
```
#include <stdio.h>
void fun(int a, int b, int *c)
  { a-=b;                          // tu a, b, c są lokalnymi zmiennymi
    *c=a*a;
  }

void main(void)
  { int i=7, j=3, k;
    fun(i ,j, &k);
    printf("i=%d  j=%d  k=%d.", i, j, k);
  }
```

Wewnątrz funkcji zmniejszeniu z 7 do 4 (*a–=b;*) ulega tylko lokalna zmienna *a*, co pozostaje bez wpływu na wartość zmiennej *i*. Wskaźnik *c* wskazuje na zmienną *k*. Tak więc wynikiem wyrażenia **c* jest wskazywana zmienna *k*.

Jeśli parametrem i argumentem funkcji jest nazwa tablicy, to funkcja ma dostęp do elementów tej tablicy i może zapisywać do nich wartości. Jest tak dlatego, że nazwa tablicy jest stałą wskazującą na pierwszy element tablicy.

Na przykład poniższa funkcja wczyta dane bezpośrednio do tej tablicy, której nazwa zostanie wstawiona jako argument wywołania. Zostało to zilustrowane na przykładzie wprowadzania tablicy *X* w funkcji *main* na rys. 5.3



Rys. 5.3. Przekazanie tablicy do funkcji

```
void czytaj_tab(int n, double tab[])
  { int i;
    for(i=0; i<n; i++) scanf("%lf", &tab[i]);
  }
```

Pytania i zadania

5.2. Napisz bez używania zmiennych globalnych funkcję, która obliczy pole powierzchni i objętość kuli o danym promieniu R. Napisz przykładowe wywołanie tej funkcji.

5.3. Napisz bez używania zmiennych globalnych i bez jawnego definiowania zmiennych lokalnych funkcję, która obliczy sumę C wektorów (tablic) A oraz B. Elementami tablicy C są liczby $c_i = a_i + b_i$.

5.4. Napisz bez używania zmiennych globalnych funkcję, która zamieni miejscami wartości dwu swoich argumentów, gdy te argumenty są typu:
a) *int* lub *float*, b) wskazującego *int* lub *float*, c) dowolnego.

5.5. Napisz funkcję, która dla N-kąta foremnego wpisanego w okrąg o promieniu R da w wyniku: a) pole wielokąta, b) bok wielokąta.

5.3. Rezultat funkcji, instrukcja return

Funkcje, które nie zwracają wartości, są definiowane jako funkcje typu *void*. Jeśli wewnątrz funkcji typu *void* jest instrukcja **return;**, to pomiędzy nią a średnikiem może być tylko wyrażenie puste.

Na przykład

```
void czytaj_tab(int n, float tab[])
  { int i;
    if(n<1) return;
    for(i=0; i<n; i++) scanf("%f", &tab[i]);
  }
```

Funkcje zwracające wartość muszą mieć zdefiniowany typ lub są domyślnie typu *int*. Funkcje te muszą zawierać instrukcję **return** *wyrażenie***;**. Wartość tego wyrażenia jest zwracana z funkcji pod jej nazwą.

Jeśli na przykład funkcja

```
double srednia(int n, float tab[])
  { int i;
    float s;
    for(i=0, s=0.0; i<n; i++) s+=tab[i];
```

```
  return s/n;                          // można pisać return(s/n);
}
```

zostanie wywołana w instrukcji ***Sr = srednia(N, X);***, to obliczy ona średnią arytmetyczną elementów tablicy *X* i podstawi tę średnią pod zmienną *Sr*.

Pytania i zadania

5.6. Napisz funkcję, która dla danych przyprostokątnych obliczy długość przeciwprostokątnej trójkąta.

5.7. Napisz funkcję, która w danej tablicy liczb całkowitych znajdzie ostatnią liczbę o zadanej wartości i zwróci w wyniku jej indeks lub –1, gdy podanej liczby nie ma w tablicy. Nie używaj zmiennych globalnych ani nie definiuj jawnie żadnych zmiennych lokalnych.

5.8. Zmodyfikuj funkcję z zadania 5.7 tak, aby jej wynikiem było wskazanie znalezionej liczby lub wskazanie puste *NULL*, gdy liczby nie znaleziono.

5.9. Napisz funkcję, która w danym tekście znajdzie pierwsze wystąpienie zadanego znaku i da w wyniku wskazanie tego znaku lub wskazanie puste *NULL*, gdy nie znaleziono tego znaku w tekście.

5.10. Napisz funkcję, która da w wyniku sumę elementów tablicy typu *float* i użyj tej funkcji w przykładowej funkcji *srednia*.

5.4. Funkcje rekurencyjne

Funkcja rekurencyjna, to funkcja, która wywołuje samą siebie.

W prawidłowo zdefiniowanej funkcji rekurencyjnej wywołanie własne jest tak uwarunkowane, aby istniała gwarancja zakończenia rekurencji i wyjścia z funkcji. Typowym przykładem jest funkcja obliczania – $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ ze wzoru rekurencyjnego

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

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

Zauważmy, że funkcja rekurencyjna jest tu prostsza od funkcji, która obliczałaby $n!$ bezpośrednio, mnożąc w pętli liczby od 1 do n. Kolejne wywołanie z argumentem pomniejszonym o 1 gwarantuje tu, że warunek $n > 1$ przestanie kiedyś być spełniony. W wielu przypadkach algorytm rekurencyjny jest prostszy od algorytmu bezpośredniego.

Przykład 5.3

Algorytm sortowania można ująć następująco.

1. Jeśli brak jest elementu do pary zakończ algorytm.
2. Jeśli para kolejnych elementów nie zachowuje porządku to zamień elementy tej pary miejscami, a następnie sortuj parę poprzednią, jeśli taka istnieje.
3. Sortuj od następnej pary elementów.

Przykładowa rekurencyjna funkcja sortująca liczby.

```
void sort(int i, int N, int X[])
  { int y;
    if(i==N) return;
    if(X[i-1]>X[i])
      { y=X[i]; X[i]=X[i-1]; X[i-1]=y;
        if(i>1) sort(i-1,i, X);
      }
    if(N==0) return;
    sort(i+1, N, X);
  }
```

Do posortowania tablicy X zawierającej n liczb całkowitych wystarczy instrukcja

```
sort(1, n, X);
```

Przykład 5.4

Rekurencyjną funkcję obliczania sumy $S_n = a_0 + a_1 + ... + a_{n-1} + a_n$ przez S_{n-1} można zapisać w postaci

$$S_n = \begin{cases} a_0 & \text{dla } n = 0 \\ S_{n-1} + a_n & \text{dla } n > 0 \end{cases}$$

i obliczyć za pomocą funkcji

```
double suma(int n, double a[])
  {if(n<1) return(a[0]); else return (suma(n-1, a)+a[n]); }
```

Pytania i zadania

5.11. Wartość wielomianu $W_n(x) = a_0 x^n + a_1 x^{n-1} + ... + a_{n-1} x + a_n$ oblicza się ze wzoru $W_n(x) = (...(a_0 x + a_1)x + ... + a_{n-1}) x + a_n$, który można przedstawić w postaci rekurencyjnej

$$W_n(x) = \begin{cases} a_0 & \text{dla } n = 0 \\ xW_{n-1}(x) + a_n & \text{dla } n > 0 \end{cases}$$

Napisz rekurencyjną funkcję, która da w wyniku wartość wielomianu.

5.12. Dwa teksty są różne, jeśli zaczynają się różnymi znakami (tekst pusty zaczyna się znakiem zerowym '\0'). Dwa puste teksty są jednakowe. Użyj tych stwierdzeń do napisania funkcji *tekstpor*(*char *p, char *q*), która da w wyniku 0, gdy teksty wskazywane przez *p* oraz *q* są jednakowe lub 1, gdy nie są.

5.5. Parametry funkcji main

Funkcja *main* może być typu *int* albo typu *void* i może wystąpić w dwóch postaciach:
– jako funkcja bezparametrowa: **`main(void)`**
– jako funkcja dwuparametrowa: **`main(int argc, char *argv[])`**

Parametry funkcji *main* są kojarzone z argumentami wywołania programu:
– **argc** otrzymuje wartość liczby argumentów wywołania programu (*argc* ### 1),
– **argv** jest tablicą wskaźników do tekstów argumentów.

Przyjmuje się, że *argv*[0] wskazuje na nazwę programu (razem ze ścieżką dostępu) oraz, że *argv*[*argc*]==*NULL*. Jeśli np. program *kopiuj.exe* umieszczony w katalogu *C:\programy* wywołamy poleceniem *kopiuj dane*1 *dane*2, to *argv*[0], *argv*[1], *argv*[2] wskazują kolejno na teksty: *"C:\PROGRAMY\KOPIUJ.EXE"*, *"dane*1*"*, *"dane*2*"*, parametr *argc*=3, a *argv*[3]=*NULL*.

Przykład 5.5

Poniższy program wyprowadzi liczbę i parametry programu.

```
#include <stdio.h>
void main(int argc, char *argv[])
  { int i;
    printf("\n\nProgram %s ma %d parametrow:\n",
                                          argv[0], argc-1);
    for(i=1; i<argc; i++) printf("   %s\n", argv[i]);
  }
```

Pierwsze dwa znaki tekstu wskazywanego przez *argv*[0] to nazwa napędu dyskowego i znak dwukropka. Właściwa nazwa programu wskazywana jest przez wyrażenie

```
strrchr(argv[0], '\\')+1.
```

Pomiędzy znakami wskazywanymi przez *argv*[0]+2 a *strrchr*(*argv*[0], *'\\'*) jest ścieżka do katalogu z uruchomionym programem.

Przykład 5.6

Program, w którym można podać nazwę **pliku** danych w parametrze wywołania.

```
#include <stdio.h>
void main(int argc,char *argv[])
  { char buf[80];
    if(argc>1) strncpy(buf, argv[1], 79);
      else
        {  printf("\nPodaj nazwe pliku z danymi: ");
           scanf("%79s", buf)
        }
    . . . . . .
  }
```

Do najczęstszych błędów związanych z konstruowaniem i używaniem funkcji należą:

- Złe przekazanie wartości z funkcji przez parametr – użycie zmiennej zamiast wskazania do niej.
- Wpisanie tekstu w obszar parametru wywołania programu (np. *strcpy*(*argv*[0], "*A*:*DANE*.1");)

Często spotykane złe nawyki to:

- Nadużywanie zmiennych globalnych oraz unikanie funkcji z parametrami.
- Unikanie algorytmów rekurencyjnych.

Pytania i zadania

5.13. Napisz program, który doda do siebie dowolnie wiele liczb podanych w argumentach wywołania. Do konwersji tekstu na liczbę użyj funkcji *atof*(*char**) z *stdlib.h*. Argumenty błędne potraktuj jak zera.

5.14. Napisz program, który obliczy wartość średnią liczb podanych w argumentach wywołania programu zakładając, że:
   a) ilość uśrednianych liczb wynika z ilości argumentów,
   b) pierwszy argument podaje ilość liczb, a jeśli brakuje argumentów, to program prosi o wprowadzenie brakujących liczb.

Zadania laboratoryjne

5.15. Program rozwiązywania równania kwadratowego z przykładu 1.2 (rozdz. 1) podziel na funkcje: wczytywania danych, obliczania pierwiastków, wyprowadzania wyników.

5.16. W programie z przykładu 1.3 (rozdz. 1) zmień funkcję wczytywania tablicy liczb. Przyjmij, że funkcja otrzymuje maksymalny rozmiar tablicy, a ilość liczb wczytanych zwraca jako wynik.

5.17. Do programu z przykładu 1.3 (rozdz. 1) dopisz funkcję, która da w wyniku sumę elementów tablicy typu *float* i użyj tej funkcji w przykładowej funkcji *srednia*.

5.18. Opracuj i uruchom program testujący funkcję opracowaną do zadania 5.3. Zapoznaj się praktycznie z przekazywaniem wartości do funkcji i z wyprowadzaniem wyników z funkcji.

5.19. Opracuj i uruchom program testujący jedną z funkcji opracowaną do zadania 5.4. Zapoznaj się praktycznie z przekazywaniem wartości do funkcji i z wyprowadzaniem wyników z funkcji.

5.20. Uruchom program z opracowaną do zadania 5.7 funkcją o wyniku całkowitym. Zmodyfikuj tę funkcję tak, aby jej wynikiem było wskazanie znalezionej liczby lub wskazanie puste *NULL*.

5.21. Sprawdź praktycznie kilka algorytmów rekurencyjnych. Napisz program i przetestuj algorytm rekurencyjny opracowany do zadania 5.11.

5.22. Sprawdź praktycznie kilka algorytmów rekurencyjnych. Napisz program i przetestuj algorytm rekurencyjny opracowany do zadania 5.12.

5.23. Napisz program z rekurencyjną funkcją obliczającą z trójkąta Pascala współczynniki $A_n[k]$, (k = 0, 1, ..., n) rozwinięcia dwumianu $(x+y)^2$. Wykorzystaj, że $A_n[0] = A_n[n] = 1$ oraz $A_n[k] = A_{n-1}[k] + A_{n-1}[k-1]$ dla $k = 1, 2, ..., n-1$.

5.24. Napisz i uruchom program, który wypisze, na jakim dysku i w jakim katalogu jest on zainstalowany.

5.25. Uruchom i przetestuj programy opracowane do zadań 5.13 i 5.14 akceptujące argumenty wywołania.

6. Wskaźniki

Wskaźnik jest zmienną, która zawiera programowy adres innej zmiennej albo funkcji.

Jeśli np. *x* jest zmienną typu *Typ*, a *px* jest wskaźnikiem typu *Typ**, to *px* może wskazywać na *x*. Jednoargumentowy operator & podaje wskaźnik do swojego argumentu. Tak więc wyrażenie *&x* jest stałą wskazującą na zmienną *x*. Pierwsza z kolejnych instrukcji nadaje wskaźnikowi *px* wskazanie zmiennej *x*. Druga instrukcja nadaje zmiennej *x* wartość 21 ponieważ wyrażenie **px* jest zmienną *x*.

```
px=&x;
*px=21;
```

Kolejne dwie instrukcje zwiększają *x* o 1, natomiast trzecia instrukcja zwiększa wskaźnik *px* o 1, ponieważ priorytet następnikowej operacji ++ jest większy niż priorytet operacji wyłuskania *.

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

Za pomocą wskaźnika można przekazać do funkcji programowy adres (wskazanie) zmiennej oraz wskazanie funkcji.

6.1. Definiowanie wskaźników

W definicji wskaźnik jest opisany przekształceniem do definiowanego typu za pomocą operatorów: wyłuskania *, indeksowania [ ] lub wywołania funkcji ( ).

Przykłady definicji wskaźników różnych typów:

`float *k;`	**k** – wskaźnik zmiennej typu *float* (bo **k* jest typu *float*)
`int **p;`	**p** – wskaźnik wskaźnika zmiennej typu *int* (***p* jest typu *int*)
`char *L[5];`	**L** – tablica pięciu wskaźników zmiennych typu *char*
`long (*N)[12];`	**N** – wskaźnik tablicy 12 zmiennych typu *long*
`int *(*M)[8];`	**M** – wskaźnik tablicy ośmiu wskaźników zmiennych typu *int*
`double (*f)(void);`	**f** – wskaźnik funkcji bezparametrowej o wartości typu *double*
`float *(*g)(void);`	**g** – wskaźnik funkcji zwracającej wskaźnik zmiennej typu *float*
`int (*(*h)(void))[4];`	**h** – wskaźnik funkcji zwracającej wskaźnik tablicy czterech elementów typu *int*
`float (*(*r)[6])(int);`	**r** – wskaźnik tablicy sześciu wskaźników funkcji o argumencie typu *int* oraz o wartościach typu *float*

Nazwa typu definiowanej zmiennej powstaje przez opuszczenie tej zmiennej w jej definicji. Tak więc

k	jest typu	`float *`
p	jest typu	`int **`
L	jest typu	`char *[5]`
N	jest typu	`long (*)[12]`
M	jest typu	`int *(*)[8]`
f	jest typu	`double (*)(void)`
g	jest typu	`float *(*)(void)`
h	jest typu	`int (*(*)(void))[4]`
r	jest typu	`float (*(*)[6])(int)`

Definicja

```
double
**x,*y[10],(*a)[6],*(*b)[4],(*c[12])(),(*(*d[3])())[25];
```

definiuje wskaźnik do wskaźnika zmiennej typu *double*, tablicę dziesięciu wskaźników itd.

Typ **void** jest typem pustym. Wskaźniki definiowane jako *void* nie wskazują obiektów żadnego typu i dlatego na tych wskaźnikach nie można wykonywać operacji arytmetycznych.

W definicjach można inicjować wskaźniki, stosując takie same zasady jak przy inicjowaniu innych zmiennych, np.:

```
int i, j, k, *q=&k, *R[5]={&i, &j, &k};
char *ps="Tekst wskazywany przez ps";
char *PORY[]={"Wiosna","Lato","Jesien","Zima"};
double (*F[])(double)={sin, cos, tan, sqrt, log, log10};
```

Zdefiniowany wskaźnik *q* wskazuje na zmienną *k*. Instrukcja **q*=7; da na przykład taki sam efekt jak *k*=7;.

Tablica *R* ma pięć wskaźników, z których pierwsze trzy wskazują kolejno *i*, *j*, *k*, natomiast pozostałe dwa są zerowe (równe *NULL*). Np. instrukcja **R*[1]=3; podstawia 3 pod *j*.

Wskaźnik *ps* wskazuje na pierwszy znak tekstu. Tak więc na przykład instrukcja *printf* ("\n%s\n", *ps*); wydrukuje wskazywany tekst. Do wskaźnika można dodawać liczby całkowite. Tak więc np. *ps*+2 wskazuje na literę *x*, czyli wyrażenie *(*ps*+2) tożsame z wyrażeniem *ps*[2] jest równe literze *x*, a właściwie kodowi ASCII litery *x*.

Tablica *PORY* jest tablicą czterech wskaźników do wymienionych tekstów. Np. *PORY*[1] wskazuje na tekst „Lato" podobnie jak wyrażenie *PORY*+1. Wynikiem wyrażenia *PORY*[1][2] jest znak 't'.

Lista inicjacyjna tablicy *F* zawiera nazwy funkcji. Nazwa funkcji jest stałą wskaźnikową do funkcji. Tak więc tablica *F* zawiera sześć wskaźników do funkcji matematycznych o argumencie i wyniku typu *double*. Obliczenie pierwiastka można np. wykonać za pomocą instrukcji *y*=*F* [3](*x*);

Pytania i zadania

6.1. Nazwij zdefiniowane zmienne i podaj ich typy

a) `float *x;`
b) `int *L[20];`
c) `double (*V)[20];`
d) `char *(*T)[16];`
e) `char *(*V)(char*);`
f) `short (**S)[40];`
g) `long (*L)(int(*)[2])`
h) `char (*H)[4][32];`
i) `double (*M[20])(int*);`
j) `long **(**J)[80];`
k `float *(*K[5])[6];`
l) `int (*(*L[8])())[2][12];`

6.2. Zainicjuj definiowane zmienne przykładowymi wartościami. Tam gdzie należy, uzupełnij definicje rozmiarami tablic

a) `float *a;`
b) `char *b;`
c) `float (*c)[];`
d) `char *d[3];`
e) `int *e[];`
f) `double (*f)(double)`
g) `char *(*g)[];`
h) `long **h;`
i) `double *(*i)[];`
j) `int (*j[2])[];`
k) `char **k;`
l) `float (*L)[][3];`

6.3. Wskaż, które wymiary tablic nie są potrzebne, gdy do definicji zostaną dodane listy inicjacyjne

a) `int *a[5];`
b) `char (*b)[6];`
c) `double (*c)[20][15];`
d) `float *(*d)[30];`
e) `int *(*e[5])[8];`
f) `char (*f[3])()[5];`

6.2. Wskaźniki, adresy i modele pamięci

Wskaźniki mogą być dalekie – czterobajtowe, złożone z **segmentu** i z **offsetu** lub bliskie – dwubajtowe, złożone tylko z **offsetu**. Wskaźniki bliskie mają wspólny segment.

pełny wskaźnik (czterobajtowy) = segment : offset
wskaźnik dwubajtowy = offset (adresuje do 64 KB)
adres fizyczny = 16·segment + offset (typ *long*)

Są dwa rodzaje wskaźników: wskaźniki danych i wskaźniki kodu.
Wskaźniki danych mogą wskazywać na zmienne.
Wskaźniki kodu mogą wskazywać na funkcje.

Domyślną reprezentację wskaźników obu rodzajów nazywamy **modelem pamięci**. Wykaz modeli pamięci zawiera tabela 6.1.

Tabela 6.1. Modele pamięci

Model	Ograniczenia	Postać wskaźnika	
		Danych	Kodu
TINY	Dane + Kod < 64 KB	offset	offset
SMALL	Dane < 64 KB, Kod < 64 KB	offset	offset
MEDIUM	Dane < 64 KB,	offset	segment : offset
COMPACT	Kod < 64 KB	segment : offset	offset
LARGE		segment : offset	segment : offset
HUGE		segment : offset	segment : offset

W modelu HUGE wskaźniki są normalizowane, co wydłuża czas obliczeń.

Znormalizowany wskaźnik ma offset z przedziału 0 ### *offset* ### 15.

Niezależnie od modelu pamięci wskaźniki definiowane jako:
– **far** (np. *int far* **p*;) są zawsze czterobajtowe,
– **near** (np. *int near* **q*;) są zawsze dwubajtowe,
– **huge** (np. *int huge* **r*;) są zawsze czterobajtowe znormalizowane.

Wartości segmentu i offsetu mogą być przechowywane w zmiennych typu *unsigned*. Poniższe funkcje przekształcają czterobajtowe wskaźniki na segment i offset oraz odwrotnie.

```
unsigned FP_OFF(void far *farptr)
unsigned FP_SEG(void far *farptr)
void far *MK_FP(unsigned segment, unsigned offset)
```

Wartości wskaźników można wyprowadzić konwersją %*p* w postaci heksagonalnych liczb o wartościach segmentu i offsetu. Na przykład

```
printf("%p", ptr);
```

wyprowadza wskaźnik *ptr* w postaci *ssss:oooo* (*segment*:*offset*) albo *oooo* (*offset*), zależnie, czy *ptr* jest wskaźnikiem cztero- czy dwubajtowym. W pamięci pierwsze dwa bajty wskaźnika to bajty offsetu.

Pytania i zadania

6.4. Jakie są fizyczne adresy obiektów wskazywanych przez wskaźniki?
a) 0x0A00:0x00B8 b) 0x0001:0x0002 c) 0xA0BC:0x0044

6.5. Unormuj wskaźniki
a) 0x0A00:0x00F4 b) 0x0000:0xA482 c) 0x8800:0x0400

6.6. Używając funkcji MK_FP skonstruuj wskaźniki do obiektów umieszczonych pod adresami
a) 0xF8A00 b) 0xC8F46 c) 0x100

6.7. Napisz funkcję, która dla wskaźnika typu *void far** obliczy wskazywany adres.

6.8. Napisz funkcję, która dla danego adresu fizycznego wyznaczy czterobajtowy wskaźnik.

6.3. Arytmetyka na wskaźnikach

Na wskaźnikach typu **innego niż void** można wykonywać następujące operacje:
– dodać lub odjąć liczbę całkowitą od wskaźnika,
– odjąć od siebie dwa wskaźniki tego samego typu,
– porównać lub przyrównać wskaźniki tego samego typu,
– przyrównać wskaźnik do wskaźnika pustego *NULL*.

Operacje arytmetyczne +, –, +=, –=, ++, -- oraz operacje porównania <, <=, >, >= wykonywane są tylko na offsecie.

Operacje przyrównania ==, != wykonywane są na całym wskaźniku.

Dodanie (odjęcie) liczby całkowitej *n* do wskaźnika *px* wyznacza wskazanie do *n*-tego obiektu za (przed) obiektem wskazywanym przez *px*.

Jeżeli są zdefiniowane następujące wskaźniki:

```
double (*Q)[16], (*P)[80], *pB, **pA;
```

to

Q+1 wskazuje kolejną tablicę, czyli o 16 obiektów typu *double* dalej niż *Q*,
P+1 wskazuje kolejną tablicę, czyli o 80 obiektów typu *double* dalej niż *P*,
P[1] wskazuje element *P*[1][0] (początkowy element tablicy *P*[1]),
P[1]+7 wskazuje element *P*[1][7] (7 obiektów typu double dalej niż *P*[1]),
pB+1 wskazuje o jeden obiekt typu double dalej niż *pB*, czyli na *pB*[1],
***pA+1** wskazuje o jeden obiekt typu double dalej niż **pA*, czyli na (**pA*)[1],
pA+1 wskazuje o jeden obiekt typu double* dalej niż *pA*, czyli na *pA*[1].

O ile *pB* wskazuje na liczby typu double, to *pA* wskazuje na wskaźniki liczb typu *double*. Wynikiem wyrażenia *pA*[1] jest więc wskazanie na obiekt typu *double*.

Wskaźniki *Q* i *P* różnią się arytmetyką, to znaczy liczbą bajtów, o którą wyrażenia *Q*+1 oraz *P*+1 przesuwają wskazania**.**
Mimo że wskaźniki P oraz *P*[0] wskazują na ten sam bajt, wskazują na różne obiekty**. Różnią się zatem typami**, bo *P*+1 wskaże następną tablicę, podczas gdy *P*[0]+1 wskaże następną liczbę w tablicy**.**

Różnica dwu wskaźników jednakowego typu wyznacza liczbę całkowitą obiektów między tymi wskazaniami.

Poniższe wyrażenie da w wyniku liczbę 20 typu *int* niezależnie od typu tablicy *A*.

```
&A[20] - A
```

Porównanie i przyrównanie dwu wskaźników ma sens tylko wtedy, gdy te wskaźniki wskazują na elementy tej samej tablicy.
Wynikiem porównania dwu wskaźników jest wynik porównania ich offsetów. Offsety te są traktowane jako liczby typu *unsigned*.
Dwa wskaźniki są równe tylko wtedy, gdy mają równe segmenty i równe offsety.

Na przykład wskaźniki 0xA000:0x0020 (segment=0xA000, offset=0x0020) oraz 0xA002:0x0000 są różne, ale wskazują na ten sam obiekt umieszczony pod adresem 0xA0020. W modelu *HUGE* wskaźniki są normalizowane. Tak więc przykładowy

wskaźnik 0xA000:0x0020 zostanie automatycznie przekształcony do postaci 0xA002:0x0000. Dzięki normalizacji różne wskaźniki zawsze wskazują różne obiekty. W modelu *HUGE* wszystkie operacje wykonywane są na całych wskaźnikach.

Operacje następnikowe ++ -- mają wyższy priorytet od operacji wyłuskania *, natomiast operacje poprzednikowe ++ -- i operacja wyłuskania * mają równy priorytet, ale są wykonywane od prawej do lewej. Tak więc np. **s*++ zwiększa wartość *s*, nie tego zaś na co *s* wskazuje, tak jak czyni *(*s*++). Wynikiem jest wartość wskazywana przed inkrementacją wskaźnika *s*. Wyrażenie *++*s* zwiększy *s*, a następnie da wartość wskazywaną przez zwiększone *s*. Aby inkrementować to, co *s* wskazuje, należy użyć wyrażenia ++**s* lub (**s*)++.

Przykłady

Poniższa funkcja kopiuje tekst wskazywany przez *t* do bufora wskazywanego przez *s*.

```
void strcpy(char *s, char *t)
  {while(*s++=*t++);
  }
```

Funkcja *strcmp* porównuje dwa teksty i zwraca zero, gdy teksty są jednakowe lub różnicę kodów (**s*–**t*) pierwszej napotkanej pary nierównych znaków.

```
int strcmp(char *s, char *t)
  {for(;*s==*t;s++,t++) if(*s=='\0') break;
    return(*s-*t);
  }
```

Funkcja *strlen* oblicza długość tekstu. Zwracana różnica (*p*–*s*) jest równa liczbie znaków między znakiem końca ('\0') a początkiem tekstu.

```
int strlen(char *s)
  {char *p=s;
    while(*p!='\0') p++;
            // powyższą linię można uprościć do postaci:   while(*p) p++;
    return(p-s);
  }
```

Pytania i zadania

6.9. Jaka będzie zawartość tablicy *X* po wykonaniu instrukcji

```
a) for(i=0, q=X; i<4; i++, q+=2) {*q=1; *(q+1)=2;}
b) for(q=X+5; q>=X; q--) {*q--=1; *q=2;}
c) for(q=X; q-X<6; q++) *q=q-&X[2];
```

6.10. Podaj wartości wyrażeń, jeśli *q* = *&X*[3]

```
a) q-X              b) q!=X             c) q<X
d) q<=&X[3]         e) &X[3]-&X[1]      f) q+1==&X[4]
```

6.11. Jaką wartość otrzyma zmienna *k*, jeśli zdefiniowano

```
     int k, *j, N[9];     long *L;    char *p, *q;
a) j=&N[5];                  b) p=(char*)&N[5];
   k=j-N;                       k=p-(char*)N;
c) j=N+3;                    d) p=(char*)(N+3);
   k=j-N;                       k=p-(char*)N;
e) L=(long*)(N+4);           f) L=(long*)N+4;
   k=L-(long*)N;                k=(char*)L-(char*)N;
```

6.12. Wskaźniki *p* i *q* wskazują odpowiednio na początkowy i końcowy element pewnej tablicy. Napisz wyrażenie, które da w wyniku
a) liczbę elementów tablicy,
b) liczbę bajtów zajmowanych przez tablicę,
c) wskazanie środkowego elementu tablicy.

6.13. Wskaźnik *p* wskazuje na tekst. Napisz instrukcje, które
a) obliczą ile jest w tym tekście liter *a* lub *A*,
b) zamienią wszystkie spacje na znaki nowej linii,
c) przesuną wskazanie *p* na początek następnego wyrazu (lub na koniec tekstu).

6.4. Wskaźniki i tablice

W języku C nazwa tablicy jest stałą wskaźnikową, wskazującą na początkowy element tablicy.

Operator indeksacji można wyrazić za pomocą dodawania i wyłuskania, np. jeśli jedna ze zmiennych *A, n* jest typu wskaźnikowego, a druga typu całkowitego, to zachodzi równość

```
A[n] == *(A+n) .
```

Jeśli zdefiniujemy np. *int* K[10];, to wyrażenie & K[0] ma wartość K. Podobnie *K daje K[0]. Analogicznie dla *i*-tego elementu & K[i] ma wartość $K+i$, natomiast wyrażenie *($K+i$) daje w wyniku K[i].

Stała wskaźnikowa K jest tu typu *int**, ponieważ elementy tablicy K są typu *int*. Wyrażenie $K+i$ należy rozumieć jako zwiększenie wskazania tak, aby wskazywało *i*-ty obiekt (tutaj typu *int*) za obiektem wskazywanym przez K.

Zdefiniujmy

```
char *r,P[]="Tekst w tablicy P",*q="Tekst wskazywany przez q";
```

Nazwa tablicy P wskazuje na początkowy element czyli na literę T. Tak więc $r=P+2$ wskazuje na literę z, czyli $*r$ jest równe literze z. Wyrażenia $P[2]$ i $*(P+2)$ są równoważne, bo $P+2$ wskazuje dwa obiekty (tu dwa znaki) dalej niż P.

W przypadku tablic wielowymiarowych mamy faktycznie do czynienia z jednowymiarowymi tablicami, których elementami są tablice.

Na przykład tablica dwuwymiarowa jest tablicą wierszy, a każdy wiersz jest tablicą liczb. Zdefiniujmy

```
double A[7][4];
```

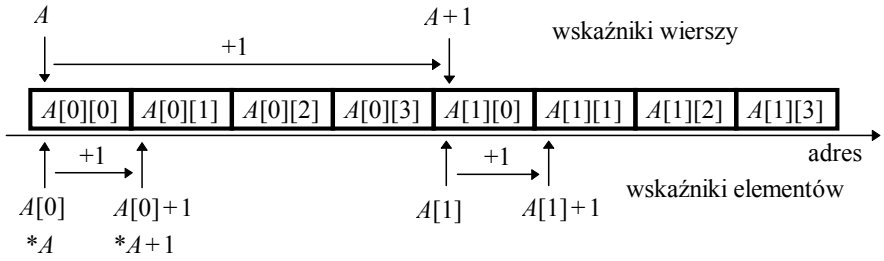
Nazwa tablicy A jest stałą wskazującą na swój początkowy element $A[0]$, czyli na pierwszy wiersz tablicy $A[7][4]$. Zatem $\&A[0]$ jest równe A. Podobnie $A+i$ wskazuje na $A[i]$, czyli na i-ty wiersz tablicy A. Dodanie i do A przesuwa wskazanie o $4i$ obiektów typu *double*, czyli o i wierszy dalej. Stała A typu *double*[7][4] jest zatem zgodna z typem *double*(*)[4].

Wyrażenie $A[i]$ jest nazwą wiersza, czyli tablicy czterech liczb typu *double*. Tak więc $A[i]$ wskazuje na $A[i][0]$, natomiast $A[i]+j$ wskazuje na $A[i][j]$. W szczególności $A[0]$ wskazuje na $A[0][0]$. Zauważmy, że A wskazuje na $A[0]$, a to dopiero wskazuje na element $A[0][0]$, który jest typu *double*. Chociaż $**A$ daje w wyniku element $A[0][0]$ typu *double*, typ stałej A nie jest zgodny z typem *double***, ponieważ wyrażenie $A+1$ wyznacza wskazanie przesunięte o cztery obiekty typu *double*, a nie o jeden wskaźnik typu *double**.

Poniższe wyrażenia są trójkami równoważne

```
A[0][0],          *A[0]          **A
A[0][3],          *(A[0]+3)      *(*A+3)
A[1][3],          *(A[1]+3)      *(*(A+1)+3)
A[i][j]           *(A[i]+j)      *(*(A+i)+j)
&A[i][j]          A[i]+j         *(A+i)+j
```

Zauważmy, że wskaźniki A oraz $*A$ (czyli $A[0]$) adresują ten sam bajt w pamięci operacyjnej, tzn. pierwszy bajt elementu $A[0][0]$. Są jednak one różnych typów. O ile $*A$ wskazuje na $A[0][0]$, to A wskazuje na cały wiersz czterech liczb od $A[0][0]$ do $A[0][3]$. Tak więc $*A+1$ (czyli $A[0]+1$) wskazuje następny element $A[0][1]$, natomiast $A+1$ wskazuje następny wiersz czyli tablicę czterech elementów od $A[1][0]$ do $A[1][3]$, jak pokazano na rys. 6.1.



Rys. 6.1. Interpretacja wskaźników do wiersza i do elementu tablicy dwuindeksowej

Ponieważ nazwa tablicy jest stałą wskaźnikową, to w wyrażeniach w miejsce nazwy tablicy można użyć zmiennej wskaźnikowej.

Na przykład definiując

```
double A[10][20], B[100], *Y, (*X)[20];
```

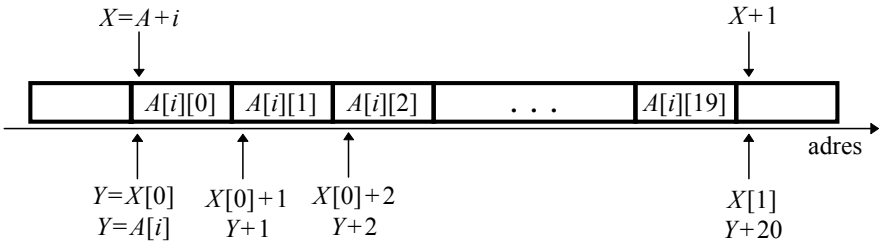
można po podstawieniu pod Y i X wskazań do odpowiednich tablic (np. $Y=B$; i $X=A$;) używać wyrażeń $Y[i]$, $X[i][j]$ oraz $X[i]$. Jeśli podstawimy:

```
Y=B+n;     X=A+m;
```

to poniższe wyrażenia są parami równoważne

```
X[i]       A[m+i]                    Y[0]       B[n]
X[0][0]    A[m][0]                   Y[i]       B[n+i]
X[i][j]    A[m+i][j]
```

Stała A wskazuje na tablicę, której elementami są wiersze zawierające po 20 elementów typu *double*. Tak więc wyrażenie $A+1$ wylicza wskazanie przesunięte o 20·*sizeof*(*double*) bajtów względem wskazania A. Zmienna X wskazuje na tablicę 20 zmiennych typu *double*. Wyrażenie $X+1$ wskazuje więc na następną tablicę czyli o 20·*sizeof*(*double*) bajtów dalej niż X. Zmienna X i stała A są zatem tego samego typu.



Rys. 6.2. Nazwa dwuindeksowej tablicy oraz zmienne wskaźnikowe

Ponieważ *A*[*i*] oraz *X*[*i*] są wskaźnikami zmiennych typu *double* (wskazują na *A*[*i*][0] oraz na *X*[*i*][0]) podobnie jak *Y*, można użyć podstawienia *Y*=*A*[*i*] oraz *Y*=*X*[*i*]. Zasadę posługiwania się wskaźnikami *A, X* oraz *Y* zilustrowano na rys. 6.2.

Ze względu na jednakową arytmetykę traktujemy nazwę tablicy *A* oraz wskaźnik *X* jako wyrażenia jednakowego typu. Istnieje jednak pewna subtelna różnica między typem stałej *A* a typem zmiennej *X*. Stała *A* jest nazwą tablicy o dziesięciu wierszach, natomiast dla wskaźnika *X* nie definiuje się liczby tablic o 20 elementach. Dlatego o ile *X* jest typu *double*(*)[20], to *A* jest typu *double*[10][20]. Operator *sizeof* traktuje nazwę tablicy i wskaźnik do jej wiersza w różny sposób, dając w wyniku rozmiar całej zdefiniowanej tablicy lub rozmiar wskaźnika. Tak wiêc

```
sizeof(A)= 10*20*sizeof(double),
sizeof(X)= sizeof(void*).
sizeof(*X)= 20*sizeof(double).
```

Zauważmy, że gdyby zdefiniowano zmienną wskaźnikową *X* jako *double* ***X*;, to jej typ nie byłby zgodny z typem stałej wskaźnikowej *A*. Zmienna *X* zostałaby zdefiniowana jako zmienna wskazująca na wskaźnik obiektu typu *double*. Tak więc wyrażenie *X*+1 wyznaczałoby wskazanie przesunięte o *sizeof*(*double**) bajtów względem wskazania *X*, czyli inne niż dla *A*.

Zauważmy, że po podstawieniu *Y*=*A*[*i*] wyrażenie *Y*[*j*] da w wyniku *A*[*i*][*j*]. Można to wykorzystać do optymalizacji pętli wewnętrznych. Na przykład zamiast obliczać sumy elementów w wierszach instrukcjami

```
for(i=0; i<10; i++)
  for(B[i]=j=0; j<20; j++) B[i]+=A[i][j];
```

można to zrobić w krótszym czasie instrukcjami

```
for(i=0; i<10; i++)
  { for(s=j=0, Y=A[i]; j<20; j++) s+=Y[j];
    B[i]=s;
  }
```

W definicji

```
char *PORY[]={"Wiosna","Lato","Jesien","Zima"};
```

tablica *PORY* ma cztery elementy. Jej elementami są wskaźniki do tekstów. Można tej tablicy używać z dwoma indeksami, tak jak się używa tablic prostokątnych, np.

PORY[0] – wskazuje na tekst „Wiosna”,
PORY[0][5] – jest znakiem *a* w tekście „Wiosna”.

Należy jednak pamiętać, że nie jest to tablica prostokątna, gdyż każdy tekst może mieć inną liczbę znaków. Tak więc o ile wyrażenie *PORY*[0][5] daje znak *a* w tekście

„Wiosna", a *PORY*[2][5] daje znak *n* w tekście „Jesien", to wyrażenia *PORY*[1][5] oraz *PORY*[3][5] nie mają sensu.

Pytania i zadania

6.14. Zapisz za pomocą notacji wskaźnikowej

```
a) A[i]          b) &B[i+1]        c) C[i][j]
d) &D[i][j]      e) *E[i]          f) F[0][0]
```

6.15. Zakładając definicję

```
char *t,T[]="Tekst 1", *q="Tekst 2", b[80], Q[80];
```

przepisać tekst z *T*[ ] do *b*[ ] oraz tekst z *q* do *Q*[ ] preferując notację a) wskaźnikową, b) indeksową.

6.16. Tablica *A* zawiera *N* liczb. Jaki będzie efekt wykonania instrukcji

```
a) for(p=A-1, i=1; i<N; i++) A[i]=p[i];
b) for(p=A-1, i=1; i<N; i++) p[i]=A[i];
c) for(p=A, i=1; i<N; i++) *p++=A[i];
```

6.17. Wyjaśnij różnice między zdefiniowanymi zmiennymi

```
a) char a[]="IBM";   b) char b[]="IBM";   c) char *c="IBM";
   char A[10]="IBM"     char *B="IBM"         char *C[]="IBM"
```

6.5. Wskaźniki a funkcje

Nazwa funkcji jest stałą wskaźnikową do tej funkcji.

Poniższe definicje na przykład kolejno definiują wskaźnik do funkcji typu *double* i o argumencie typu *double*, nadając temu wskaźnikowi wskazanie do funkcji *sin*( ) oraz tablicę dziewięciu wskaźników do funkcji trygonometrycznych i hiperbolicznych.

```
double (*f)(double)=sin;

double (*tf[])(double)=
     {sin, cos, tan, asin, acos, atan, sinh, cosh, tanh};
```

Poniższe instrukcje są tu parami równoważne instrukcjom $y = sin(x)$; $y = tan(x)$;.

```
y=(*f)(x);          oraz   y=f(x);
y=(*tf[2])(x);      oraz   y=tf[2](x);
```

Wskaźniki na funkcje można:
- przekazywać jako argumenty innym funkcjom,
- zwracać jako wynik funkcji,
- porównywać ze wskazaniem zerowym NULL,
- poddawać operacji wyłuskania.

Na wskaźnikach na funkcje nie wolno wykonywać operacji arytmetycznych.

Na przykład funkcja

```
void tablicuj(float min, float max, float d,
                              double *T, double (*f)(double))
  {double x;
    for(x=min; x<=max; x+=d) *T++=f(x);
  }
```

może tablicować dowolne funkcje typu *double*(*)(*double*) do podanej tablicy *T*, natomiast poniższa funkcja *tab* o wyniku typu *double*(*)[2] zaalokuje i wypełni tablicę *n* par liczb $x_i, f(x_i)$.

```
double (*tab (float min, float max, int n,
                                    double (*f)(double)))[2]
  { int i;
    double x, d, (*T)[2];
    if(n<2) return(NULL);
    d=(max-min)/(n-1);
    T=(double(*)[2])calloc(n, sizeof(T[0]));
    x=min;
    for(i=0; i<n; i++, x+=d)
       {T[i][0]=x; T[i][1]=f(x);}
    return(T);
  }
```

Przykładowe użycie powyższej funkcji ilustruje program

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <math.h>
double (*tf[])(double)=
       {sin, cos, tan, asin, acos, atan, sinh, cosh, tanh};
double (*tab (float min, float max, int n,
                                    double (*f)(double)))[2]
  {  .  .  .  // tu definicja funkcji tab jak wyżej
  }
```

```
void main(void)
  {int nr, N, i;
    double xmin, xmax, (*T)[2];
    clrscr();
    printf("Podaj nr: 1-sin, 2-cos, 3-tg, 4-arcsin, "
                "5-arccos, 6-arctg, 7-sh, 8-ch, 9-th");
    do scanf("%d", &nr); while (nr<1||nr>9);
    printf("\nPodaj: xmin, xmax, N : ");
    scanf("%lf%lf%d", &xmin, &xmax, &N);
    T=tab(xmin, xmax, N, tf[nr-1]);
    for(i=0; i<N; i++)
         printf("\nf(%.3lf)= %.3lf", T[i][0], T[i][1]);
    getch();
  }
```

Podobnie biblioteczna funkcja *qsort* może sortować tablice dowolnych obiektów, ponieważ jednym z jej parametrów jest wskaźnik do funkcji porównującej sortowane obiekty. Funkcja *qsort* posiada nag³ówek

```
void qsort(void *base, size_t nelem, size_t width,
                int (*fcmp)(const void *, const void *))
```

Ostatnim parametrem o nazwie *fcmp* jest wskaźnik do funkcji typu *int* o dwu argumentach typu wskaźnik. Przykładowe wywołanie *qsort* do posortowania tablicy *N* liczb wymaga zdefiniowania funkcji dającej wynik porównania sortowanych elementów w postaci liczby całkowitej < 0, $= 0$ lub > 0 i może być następujące:

```
int porownaj(const void *pa, const void *pb)
  {double a=* (double*)pa, b=* (double*)pb;
    return((b>a)-(b<a));
  }

main()
  {int N;
    double X[1000];
  . . . . . .
    qsort(X, N, sizeof(X[0]), porownaj);
  . . . . . .
  }
```

Tej samej funkcji *qsort* można użyć do posortowania tekstów w kolejności alfabetycznej. Niech tablica *P* zdefiniowana np. jako *char* **P*[1000] zawiera $N <= 1000$ wskaźników do sortowanych tekstów. Niech funkcja porównania nazywa się *stracmp*.

```
int stracmp(const void *s, const void *t)
  {char *cs =*(char**)s, *ct =*(char**)t;
    for(;toupper(*cs)==toupper(*ct);cs++,ct++)
                                    if(*cs =='\0') break;
    return(toupper(*cs)-toupper(*ct));
  }
```

Powyższą funkcję można zdefiniować prościej

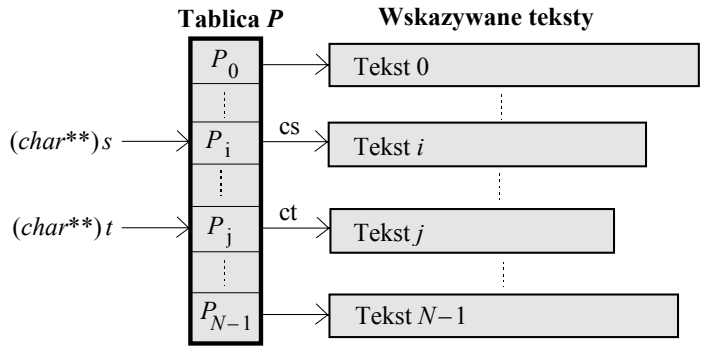
```
int stracmp(const void *s, const void *t)
  {return(stricmp(*(char**)s, *(char**)t));}
```

Instrukcja

```
qsort(P, N, sizeof(P[0]), stracmp);
```

posortuje tablicę wskaźników.

Funkcja porównująca *stracmp* otrzymuje wskaźniki do sortowanych elementów, czyli otrzymuje wskaźniki do wskaźników do tekstów. Choć formalnie parametry *s* oraz *t* są typu *const void*, to faktycznie przekazują wskaźniki typu *char***, jak pokazano na rys. 6.3. Aby wyłuskać wskaźniki do porównywanych tekstów należy najpierw rzutować zmienne *s* oraz *t* na właściwy typ czyli na *char***.



Rys. 6.3. Sortowanie tablicy wskaźników do tekstów

Pytania i zadania

6.18. Zainicjuj definiowane wskaźniki przykładowymi wartościami i użyj tych wskaźników do pośredniego wywołania przykładowych funkcji

a) *int* (**a*)( ); b) *char* *(**b*)( ); c) *char* *(**c*[ ])( );
d) *double* (**d*)( ); e) *int* (**e*[ ])( ); f) *double* (*(**f* [ ])( ))[4];

6.19. Napisz funkcję, która w danej tablicy wybierze element maksymalny w dowolnym sensie i da jako wynik: a) numer elementu, b) wskazanie do elementu.

6.20. Napisz dla funkcji *qsort* funkcję porównującą, która umożliwi posortowanie
   a) tablicy liczb całkowitych,
   b) tablicy (typu *float*[ ][4]) współrzędnych końców odcinków według ich długości,
   c) tablicy wskaźników do tablic typu *float*[4], zawierających współrzędne końców odcinków według długości tych odcinków,
   d) wierszy tablicy dwuwymiarowej o elementach typu *int* według sumy liczb w wierszu.

   Podaj przykłady wywołania funkcji *qsort*.

6.21. Jeśli funkcja $f(x)$ jest ciągła i monotoniczna w przedziale $x\#\#\#<a, b>$ oraz wartości $f(a)$ i $f(b)$ mają różne znaki, to w tym przedziale istnieje jeden pierwiastek x_0 tej funkcji ($f(x_0)=0$). Napisz funkcję, która zawężając przedział $<a, b>$, znajdzie dla dowolnej funkcji pierwiastek z zadaną dokładnością.

6.6. Alokacja pamięci

Biblioteczne funkcje przydziału pamięci to:

`void *malloc(size_t S)`	– wskazanie S bajtów,
`void *calloc(size_t N, size_t S)`	– wskazanie $N \cdot S$ bajtów,
`unsigned coreleft(void)`	– liczba wolnych bajtów,
`void free(void *ptr)`	– zwolnienie obszaru.

Typ *size_t* jest zdefiniowany jako *unsigned*. Funkcje *malloc* i *calloc* przydzielają do programu spójne obszary pamięci, nie przekraczające 64 KB i zwracają jako wynik wskazanie do przydzielonego obszaru. W przypadku, gdy alokacja nie jest możliwa (np. brak pamięci), funkcje te zwracają **wskaźnik pusty *NULL***. Funkcja *calloc* dodatkowo zeruje przydzieloną pamięć. Funkcja *free* zwraca do systemu operacyjnego przydzielony wcześniej obszar pamięci.

Fatalnym błędem jest zwracanie pamięci, która **nie** została przydzielona.

Funkcja *coreleft* daje rozmiar (do 64 KB) maksymalnego obszaru spójnego. Do przydzielania większych obszarów pamięci są odpowiedniki powyższych funkcji: *farmalloc*(*long S*), *farcalloc*(*long N*, *long S*), *long farcoreleft*(*void*) oraz do zwalniania pamięci jest funkcja *farfree*(*far void *ptr*).

Aby dynamicznie utworzyć *L* bajtowy bufor oraz tablicę *N* liczb całkowitych (typu *int*), należy zdefiniować odpowiednie wskaźniki i zainicjować je wskazaniami na zaalokowane obszary pamięci, np.

```
char *buf = (char*)malloc(L);
int  *K = (int*)calloc(N, sizeof(*K));
```

Dalej można używać wskaźników *buf* oraz *K*, tak jak gdyby były one nazwami tablic. Gdy bufor i tablica przestaną być potrzebne, należy zwolnić pobraną dla nich pamięć

```
free(buf);
free(K);
```

Poniższy przykład pokazuje funkcję, która wczytuje wiersz tekstu do odpowiednio zaalokowanego bufora. Linia tekstu jest najpierw wczytywana do dużej tablicy. Po określeniu długości tekstu alokowany jest odpowiedni bufor. Jako wynik funkcja zwraca wskazanie do tego bufora.

```
char *czytaj_wiersz(FILE *we)
{ char B[256]={'\0'}, *bb;                    // definicja tablicy i wskaźnika
  if(!fgets(B, 256, we)) return NULL;         // wczytanie wiersza do tablicy B
  bb = (char*)malloc(strlen(B)+1);            // alokacja bufora na wiersz
  if(bb) strcpy(bb, B);                       // przepisanie tekstu z tablicy do bufora
  return(bb);
}
```

A oto fragment programu, który wczyta zadaną liczbę *n* wierszy. Wskaźniki buforów zapamiętane są w dynamicznie utworzonej tablicy *P*. Organizacja tekstów przypomina pokazaną na rys. 6.3.

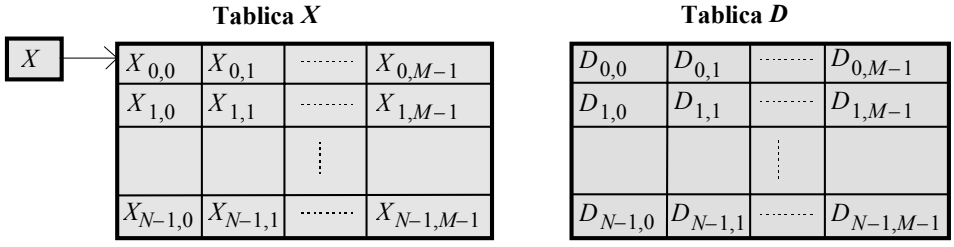
```
int i, n;
char **P;                                     // wskaźnik na tablicę wskaźników
//Tu należy wczytać lub określić liczbę wierszy (wartość zmiennej n).
P=(char**)calloc(n, sizeof(*P));              // alokacja tablicy wskaźników
if(!P) goto er1;
for(i=0; i<n; i++)
  { P[i]=czytaj_wiersz(stdin);
    if(!P[i]) {n=i; break;}                   // po błędzie skończ wczytywać
  }
/* Tu można posługiwać się tablicą tekstów. P[i] wskazuje i-ty tekst. Gdy teksty
 przestaną być potrzebne, należy zwolnić pobraną pamięć. */
for(i=n-1; i>=0; i--) free(P[i]);             // zwolnienie buforów
free(P);                                      // zwolnienie tablicy wskaźników
et1:
```

Nieco trudniejszym, ale podobnym zadaniem jest alokacja tablic dwuindeksowych.

Gdy wymiary *N*, *M* dwuwymiarowej tablicy *D* są znanymi stałymi, można ją zdefiniować jako *double D* [*N*][*M*]. Gdy wymiary tablicy są zmiennymi zależnymi od danych, to tablicę trzeba definiować z nadmiarem do maksymalnych rozmiarów lub dynamicznie alokować dla niej pamięć. Funkcje alokacji pamięci pozwalają elastycznie pobierać do programu tylko tyle pamięci, ile jest rzeczywiście potrzebne. Nie ma problemu z alokacją pamięci dla tablic jednowymiarowych. Jeśli np. dwuwymiarowa tablica *X* ma znaną liczbę *M* kolumn, to można ją potraktować jako jednowymiarową tablicę *M*-elementowych wierszy i alokować w jednym spójnym obszarze.

```
double (*X)[M];
    // wczytanie N - liczba wierszy tablicy X,
X=(double(*)[M])calloc(N, sizeof(X[0]));
if(X==NULL) goto error;
    // tu można używać wyrażeń X[i][j]
  free(X);
```

Tablica *X* zajmuje jeden obszar pamięci na $N \cdot M$ liczb (tu typu *double*) oraz pamięć na wskaźnik do tablicy, czyli pamięć na zmienną *X* (por. rys. 6.4). Zauważmy, że tu *sizeof* (*X* [0]) = $M \cdot$ *sizeof* (*double*).



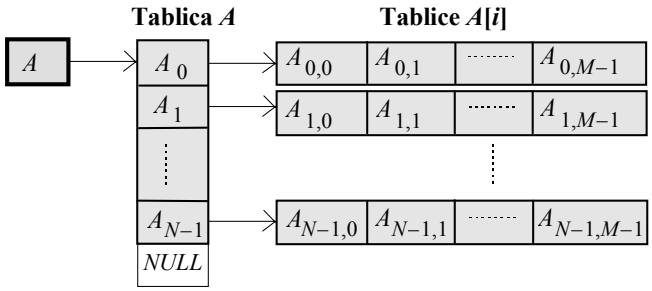
Rys. 6.4. Organizacja pamięci tablic *X* oraz *D*

Jeżeli oba wymiary *N* – liczba wierszy i *M* – liczba kolumn tablicy *A* nie są z góry znane, to aby w pełni dynamicznie zaalokować tę tablicę, należy przydzielać pamięć osobno dla każdego wiersza. Wskaźniki do wierszy należy pamiętać w dynamicznej tablicy wskaźników. Organizację pamięci tak zaalokowanej tablicy *A* pokazano na rys. 6.5.

Przykład alokacji tablic 1 i 2 wymiarowych z obsługą błędów

```
{int i, j, N, M;
 double **A,*B;
 errno=ENOMEM;
```

```
        . . . . . . . . . . . . . .
        // wczytanie rozmiarów tablic (N – liczba wierszy A, M – liczba kolumn A
i elementów B)
        . . . . . . . . . . . . . .
    B=(double*)calloc(M, sizeof(B[0]));
    if(B==NULL) goto et1;
    A=(double**)calloc(N, sizeof(A[0]));
    if(!A) goto et2;                                  // to samo co A==NULL
    for(i=0; i<N; i++)
      if((A[i]=(double*)calloc(M, sizeof(A[0][0])))==NULL)
                                                      goto et3;
        . . . . . . . . . . . . . .
        // tu można używać wyrażeń A[i][j] oraz B[i]
        . . . . . . . . . . . . . .
    errno=0;
    i=N;
       et3:  for(i--; i>=0; i--) free(A[i]);
     free(A);
       et2:  free(B);
       et1:  return(errno);
  }
```



Rys. 6.5. Organizacja pamięci tablicy zaalokowanej wierszami

Zauważmy, że A wskazuje na tablicę N wskaźników. Każdy z tych wskaźników wskazuje na inną tablicę M liczb typu *double*. Każda z N tablic mogłaby mieć inną liczbę elementów. Organizacja $N \times M$ liczb w tablicy A jest inna niż organizacja tablicy X oraz tablicy D. Tablica A zajmuje N obszarów pamięci na wiersze, jeden obszar na wskaźniki do wierszy oraz zmienną A, natomiast tablice X i D zajmują po jednym spójnym obszarze pamięci, jak pokazano na rysunku 6.4

Zauważmy, że aby zwolnić pamięć tablicy *A*, należy najpierw zwolnić obszary pamięci przydzielone wierszom tablicy, a następnie należy zwolnić pamięć przydzieloną na tablicę wskaźników.

Poniżej podano przykładowe funkcje do zwalniania i alokacji tablic prostokątnych dowolnego typu, zbudowanych w pełni dynamicznie, podobnie jak pokazane na rys. 6.5. W tym przykładzie alokowana tablica wskaźników ma $N+1$ elementów (o jeden element więcej niż wyróżniona część na rysunku 6.5). Założono tu bowiem, że ostatni wskaźnik będzie równy *NULL*, co wykorzystuje funkcja *freetab*, która zwalnia pamięć po zaalokowanej tablicy. Dzięki temu funkcja ta nie potrzebuje informacji o liczbie wierszy zwalnianej tablicy. Funkcja *alloctab* poza wymiarami prostokątnej tablicy otrzymuje rozmiar jej elementu w bajtach. Tak więc funkcja ta może alokować prostokątne tablice o elementach dowolnego typu. Podobnie jak biblioteczne funkcje *malloc* i *calloc* funkcja *alloctab* jest typu *void** i zwraca wskaźnik pusty *NULL*, jeśli nie może zaalokować całej tablicy. Funkcja *freetab* podobnie jak funkcja *free* ma jeden argument typu *void**. Gdyby do tablicy wskaźników z rys. 6.3 dodać na końcu element z pustym wskaźnikiem, jak pokazano na rys. 6.5, funkcja *freetab* mogłaby zwalniać pamięć alokowaną dla tej tablicy i dla wszystkich tekstów.

```
#include <alloc.h>

void freetab(void *AA)
{ int i;
  void **A=(void**)AA;
  if(!AA) return;
  for(i=0; A[i]; i++) free(A[i]);
  free(AA);
}

void *alloctab(int N, int M, int Size)
{ int i;
  void **A;
  if((A=(void**)calloc(N+1,sizeof(*A)))==NULL) return(NULL);
  for(i=0; i<N; i++)
    if((A[i]=calloc(M,Size))==NULL)
                        {freetab(A); return(NULL);}
  A[N]=NULL;
  return(A);
}

void main(void)
{ double **A;
  int N, M;
     // wczytanie rozmiarów tablicy A (N – liczba wierszy, M – liczba kolumn)
```

```
    A=(double**)alloctab(N, M, sizeof(**A));
    if(A==NULL) goto error;
      // tu można używać wyrażeń A[i][j]
    freetab(A);
    getch();
}
```

Do typowych błędów w używaniu wskaźników należy:
- Posługiwanie się nie zainicjowanymi wskaźnikami.
- Przesuwanie wskaźnika do elementu tablicy poza koniec tablicy.
- Inkrementacja wskaźnika wartości wskazywanej (np. *p++; zamiast (*p)++).
- Brak kontroli, czy alokacja pamięci zakończyła się sukcesem.

Złym nawykiem jest unikanie stosowania wskaźników oraz dynamicznej alokacji tablic i używanie tablic o maksymalnych rozmiarach.

Pytania i zadania

6.22. Zdefiniuj odpowiedni wskaźnik i zaalokuj pamięć na
a) bufor znakowy o pojemności *N* bajtów,
b) tablicę znakową na tekst umieszczony w tymczasowym buforze *B*[100],
c) tablicę *N* liczb typu *int*,
d) tablicę liczb typu *float* o *N* wierszach i czterech kolumnach,
e) tablicę liczb typu *long* o *N* wierszach i *M* kolumnach.

6.23. Poniższe definicje są poprawne, gdy *N* oraz *M* są stałymi całkowitymi. Zdefiniuj wskaźniki i dokonaj odpowiednich alokacji pamięci, tak aby *N* i *M* mogły być zmiennymi typu *int*

```
a) int A[N];        b) char *B[N];      c) double C[N][8];
d) long D[5][M];    e) float E[N][M];   f) char *F[N]();
```

6.24. Napisz funkcję, która dla dowolnej liczby *N* zaalokuje tablicę na *N* liczb typu *double*, wczyta te liczby i zwróci wskaźnik do tej tablicy. Uwzględnij obsługę błędów.

6.25. Napisz funkcję, która posługując się buforem na 80 znaków będzie wczytywać daną w parametrach liczbę *N* wyrazów, alokując dla nich odpowiednie bufory. Funkcja zwróci wskaźnik do zaalokowanej tablicy wskaźników, do buforów słów. Uwzględnij obsługę błędów.

6.26. Napisz funkcję, która zwolni pamięć zaalokowaną przez funkcję z zadania 6.25 zakładając, że ostatnim elementem tablicy jest zerowy wskaźnik *NULL*.

6.27. Napisz funkcję, która zaalokuje największą możliwą w chwili realizacji programu tablicę typu *long*. Wykorzystaj funkcje *farcoreleft* i *farmalloc*. Wyznaczoną liczbę elementów tej tablicy należy przekazać używając argumentu. Wynikiem funkcji ma być wskaźnik do zaalokowanej tablicy.

Zadania laboratoryjne

6.28. Napisz bez używania operatora indeksacji [ ] program, który wczyta tablicę liczb typu *double* i wybierze z tej tablicy element minimalny i maksymalny.

6.29. Napisz odpowiedni program sortowania tablicy liczb całkowitych z użyciem bibliotecznej funkcji sortowania *qsort*. Napisz odpowiednią dla *qsort* funkcję porównywania liczb całkowitych.

6.30. Napisz odpowiednie programy z użyciem funkcji sortowania *qsort* i przetestuj funkcje napisane do zadania 6.20b, c, d.

6.31. Napisz program obliczania pierwiastków funkcji metodą interpolacji liniowej (reguła falsi). Wykorzystaj funkcję opracowaną do zadania 6.21.

6.32. Napisz program z użyciem funkcji opracowanej do zadania 6.24, który z N wczytanych liczb wyprowadzi liczbę największą i najmniejszą. Uwzględnij obsługę błędów i nie zapomnij zwolnić zaalokowaną pamięć.

6.33. Napisz program z użyciem funkcji opracowanych do zadań 6.25 i 6.26, który ułoży wczytane wyrazy w kolejności alfabetycznej. Wykorzystaj funkcję *qsort*.

6.34. Napisz program, który sprawdzi i wypisze wielkość dostępnej pamięci operacyjnej. Użyj funkcji typu *far* i modelu *LARGE*. Uruchom ten program z menu platformy kompilatora, a następnie z poziomu DOSa.

6.35. Napisz program szukania liczb pierwszych dowolną metodą. Ilość szukanych liczb uzależnij od rozmiaru dostępnej pamięci operacyjnej. Użyj funkcji typu *far* i modelu *LARGE*.

6.36. Napisz program mnożenia $C=A\times B$ macierzy A o N wierszach i M kolumnach przez wektor B. Program powinien składać się z: funkcji wczytywania wymiarów macierzy A wraz z alokacją pamięci i wczytaniem elementów macierzy, funkcji alokującej i wczytującej wektor B, funkcji alokującej i obliczającej wynikowy wektor C oraz z funkcji zwalniającej pamięć przydzieloną do macierzy A.

7. Struktury danych

Struktura jest obiektem złożonym z jednej lub kilku zmiennych (składowych), zwykle różnych typów, dla wygody zgrupowanych pod jedną nazwą. W języku Pascal struktura nazywana jest rekordem. Obiekty jednakowego typu najczęściej grupuje się w tablice. Rozróżniamy dwa rodzaje obiektów strukturalnych:

Struktury – złożone z szeregowego ciągu składowych dowolnego typu. Struktury przechowują w danej chwili wszystkie swoje składowe.

Unie – zmienne mogące pomieścić obiekty różnych typów i rozmiarów, ale w danej chwili przechowujące tylko jeden obiekt (jedną składową).

Zasady posługiwania się strukturami i uniami opierają się na wspólnych regułach:

- Deklaracja struktury/unii ma postać:
  struct lub **union** *Nazwa* **{** *deklaracje składowych* **};**
- Deklaracje składowych mają postać definicji zmiennych (obiektów).
- Nazwę struktury/unii można pominąć.
- Wyboru składowych dokonuje się operatorami **.** (kropka) lub **–>** (minus, większe).
- Nazwa struktury/unii poprzedzona słowem kluczowym *struct* lub *union* (w języku C++ sama nazwa) jest nazwą typu (podobnie jak *int* czy *double*).

Za klamrą kończącą listę składowych struktury może wystąpić lista definiowanych zmiennych strukturowych.

Nazwy: zmiennej, struktury/unii i jej składowej się nie przesłaniają.

7.1. Właściwości struktur

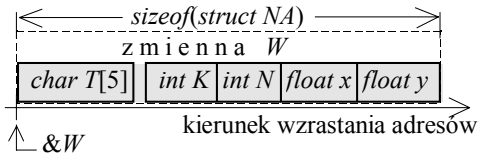
Wymienione w deklaracji **składowe struktury są umieszczane w pamięci jedna za drugą** w takiej kolejności, w jakiej je wymieniono w deklaracji. Każda składowa struktury jest umieszczana w pamięci pod adresem zgodnym z wymaganiami jej typu. W strukturze mogą zatem wystąpić miejsca puste. **Rozmiar struktury** (liczba zajmowanych bajtów) **jest równy lub większy od sumy rozmiarów wszystkich składowych.**

Zdefiniujmy przykładową strukturę o nazwie *NA*

```
struct NA
  { char T[5];          // pięciobajtowa tablica znakowa
    int K, N;           // dwie zmienne typu int
    float x, y;         // dwie zmienne typu float
  } W;                  // definiowana jest struktura W
```

Struktura *W* oraz jej komponent całkowity *K* muszą być umieszczone pod adresami parzystymi; za ostatnim elementem tablicy *T* musi więc wystąpić jednobajtowa szczelina. Struktura *NA* ma budowę pokazaną na rys. 7.1 oraz rozmiar

```
sizeof(struct NA) ### 5 + 2*sizeof(int) + 2*sizeof(float)
```



Rys. 7.1. Rozmieszczenie w pamięci składowych struktury *NA*. Zmienna *W*, jej rozmiar, zawartość i wskaźnik &*W*.

Wskaźnik do struktury i wskaźnik jej pierwszej składowej wyznaczają ten sam adres fizyczny (rys. 7.1). Choć np. &*W* wyznacza adres pierwszego bajta tablicy *T*, to jednak wskazuje na całą strukturę. Pomimo że wskaźniki &*W* i *W.T* wskazują na ten sam adres fizyczny, nie należy ich utożsamiać z następujących powodów:

- Są one różnych typów (&*W* jest typu *struct NA**, a *W.T* jest typu *char*[5]).
- Wskaźnik &*W* może być argumentem operatora wyboru –> (np. (&*W*)–>*x*), natomiast *W.T* może być argumentem operatora indeksacji [ ] (np. *W.T*[3]).
- Wskaźnik *W.T* jest wynikiem wyrażenia (&*W*)–>*T*.

Jeśli struktura posiada nazwę, to można tej nazwy używać w definicjach i deklaracjach, tak jak na przykładzie definicji struktury o nazwie *Towar*

```
struct Towar
  { char nazwa[24];         // składowa (nazwa) typu char[24]
    float cena;             // składowe (cena) typu float
    int ilosc;              // składowe (ilosc) typu int
    char jednostka[6];      // składowa (jednostka) typu char[6]
  };                        // koniec definicji struktury
. . . . . .
struct Towar chleb, bulka, maslo, *wsk; // definicja
```

Powyżej zdefiniowano trzy struktury (*chleb, bulka i maslo*) oraz jeden wskaźnik (*wsk*) typu *struct Towar**.

Inicjowanie struktur podlega takim samym prawom jak inicjowanie tablic.
- Nie wolno inicjować struktur automatycznych.
- Wartości inicjujące kolejne składowe struktury powinny być ujęte w nawiasy klamrowe { }. Lista inicjacyjna nie może zawierać więcej wartości niż struktura ma składowych. Gdy na liście inicjacyjnej jest mniej wartości niż składowych, to pozostałe składowe inicjowane są zerami.
- Jeśli składową jest obiekt złożony (tablica, struktura), to lista inicjacyjna dla tego obiektu powinna (ale nie musi) być ujęta w nawiasy klamrowe i nie musi być wtedy kompletna.

Na przykład w strukturze

```
struct Towar chleb={"Pszenny", 1.20, 150, "szt."};
```

tekst "Pszenny" zajmuje siedem elementów tablicy *nazwa*. Pozostałe elementy tej tablicy są inicjowane zerami. Składowe *cena* i *ilosc* otrzymują kolejno wartości 1.2 i 150. Tekst "szt." inicjuje tablicę *jednostka*.

Składową struktury może być inna struktura lub wskaźnik dowolnej (nawet takiej samej) struktury.

Przykład

```
struct DATA
  { short day; char month[12]; int year;};
struct OSOBA
  { char imie[16], nazwisko[24];
    struct DATA d_ur;
    struct OSOBA *ojciec, *matka;
  } P,*q=&P;
```

Rok urodzenia dają następujące wyrażenia:

```
P.d_ur.year,          q->d_ur.year,      (*q).d_ur.year
```

natomiast wskaźnikami do nazwy miesiąca urodzenia są wyrażenia

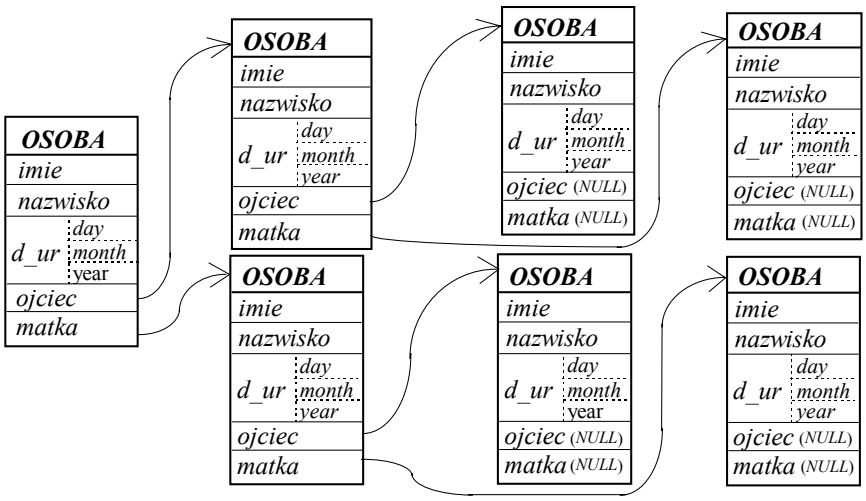
```
P.d_ur.month,         q->d_ur.month,     (*q).d_ur.month
```

Pierwszą literę miesiąca urodzenia dają wyrażenia:

```
*P.d_ur.month,        *q->d_ur.month,    *(*q).d_ur.month,
P.d_ur.month[0],      q->d_ur.month[0], (*q).d_ur.month[0]
```

Imię ojca wskazują

```
P.ojciec->imie, q->ojciec->imie,  (*q).ojciec->imie,
(*P.ojciec).imie,     (*(*q).ojciec).imie
```



Rys. 7.2. Przykładowe drzewo struktur

Rok urodzenia ojca

`P.ojciec->d_ur.year, q->ojciec->d_ur.year,` itd.

Rok urodzenia dziadka ze strony matki

`P.matka->ojciec->d_ur.year,` itd.

Rok urodzenia babki ze strony ojca

`P.ojciec->matka->d_ur.year,` itd.

Przykładowe drzewo zbudowane ze struktur *OSOBA* pokazano na rys. 7.2. Wskaźniki umieszczone w składowych *ojciec*, *matka* typu *struct OSOBA* wskazują na kolejne gałęzie drzewa. Na końcach gałęzi wskaźniki te niczego dalej nie wskazują. Pustym wskaźnikom przyjęto nadawać wartość zerową *NULL*.

Pytania i zadania

7.1. Używając operatora *sizeof* podaj minimalne rozmiary struktur

```
a)  struct {        b)  struct {        c)  struct {
      char A[4];          double x, y;        double *Z[5];
      int w; };           char *p[8];};       int k[6],*s;};
```

7.2. Zaproponuj odpowiednie struktury do zapamiętania: a) czasu z dokładnością do 1 sekundy, b) współrzędnych punktu w przestrzeni trójwymiarowej, c) współrzędnych geograficznych, d) nazwy funkcji i wskaźnika do tej funkcji, e) wskaźnika do nazwy funkcji i wskaźnika do tej funkcji,

7.3. Zdefiniuj zmienne i wskaźniki do struktur zaproponowanych w zadaniu 7.2. Napisz wyrażenia nadające przykładowe wartości tym zmiennym.

7.4. Odwołując się do obiektu struktury OSOBA bezpośrednio i przez wskazanie, napisz wyrażenia dające w wyniku: a) wskazanie imienia, b) dzień urodzenia matki, c) wskazanie nazwy miesiąca urodzenia babki ze strony ojca, d) pierwszą literę imienia i wskazanie nazwiska dziadka ze strony matki.

7.2. Struktury odwołujące się do siebie

Jeśli składową struktury jest wskaźnik do tej struktury, to mówimy, że struktura odwołuje się do siebie. Taką strukturą jest np. *struct OSOBA*, ponieważ jej składowe *ojciec* i *matka* są typu *struct OSOBA**, czyli są wskaźnikami do struktury *OSOBA*.

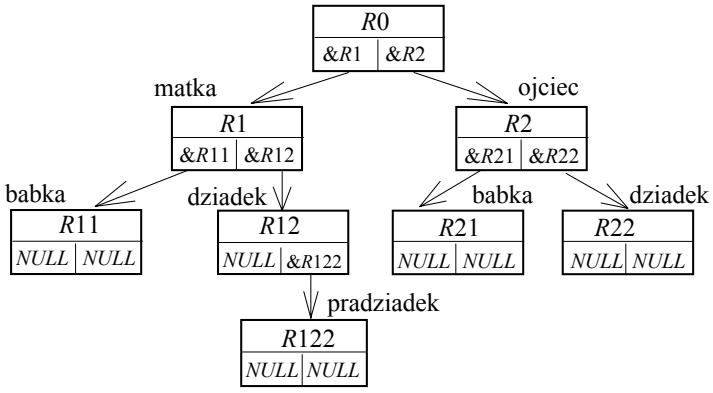
Struktury odwołujące się do siebie nadają się do tworzenia dynamicznych drzew (rys. 7.2 i rys.7.3) , stosów (rys. 7.4), łańcuchów (rys.7.5) i innych grafów.

7.2.1. Drzewa

Drzewo jest takim połączeniem struktur – węzłów drzewa – które umożliwia poruszanie się jednostronne z możliwością wyboru drogi. Przykłady drzewa pokazują rysunki 7.2 i 7.3. Dołączenie kolejnego węzła do drzewa polega na alokacji struktury węzła, nadaniu składowym struktury wartości i połączeniu nowego węzła z drzewem przez zapisanie wskaźnika nowego węzła do istniejącego węzła w miejsce wskaźnika zerowego *NULL*. Przykładowe dołączenie pradziadka w punkcie pokazanym na rys. 7.3 realizują instrukcje:

```
struct OSOBA *q, *q1;
. . . . . .
q=P.matka.ojciec;                    // q wskazuje na dziadka (ojca matki)
q1=(struct OSOBA*)malloc(sizeof(struct OSOBA));
                                     // pobranie pamięci
if(q1==NULL) goto error;             // skok po nieudanej alokacji pamięci
q->ojciec=q1;                        // włączenie struktury *q1 do drzewa
strcpy(q1->imie,"Jan");              // wpisanie imienia do struktury
strcpy(q1->nazwisko,"Kowal");        // wpisanie nazwiska do struktury
q1->d_ur.day=23;                     // wpisanie dnia urodzenia
strcpy(q1->d_ur.month,"maj");        // wpisanie miesiąca urodzenia
q1->d_ur.year=1902;                  // wpisanie roku urodzenia
q1->ojciec=q1->matka=NULL;           // zaznaczenie końca gałęzi
```

Po drzewie i po stosie można poruszać się tylko w jednym kierunku. Na przykład, jeśli wskaźnik *q* wskazuje na węzeł *R*1 (rys. 7.3), to można obliczyć



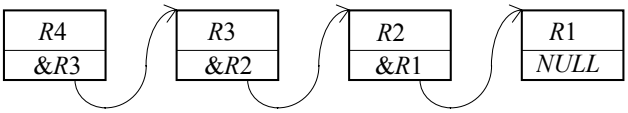
Rys. 7.3. Powiązanie struktur OSOBA w drzewo

```
q->matka                    // wskazanie na R11
q->ojciec                   // wskazanie na R12
q->ojciec->ojciec           // wskazanie na R122
```

natomiast nie można obliczyć wskaźników do pozostałych węzłów drzewa.

7.2.2. Stosy

Stos jest przypadkiem uproszczonego drzewa, w którym istnieje tylko jedna gałąź i można się poruszać tylko w jednym kierunku i jedną drogą. Do utworzenia stosu wystarczy struktura z jednym wskaźnikiem do siebie, jak pokazano na rys. 7.4.



Rys. 7.4. Powiązanie struktur w stos

Przykłady tworzenia i modyfikacji stosu

Położenie na stos

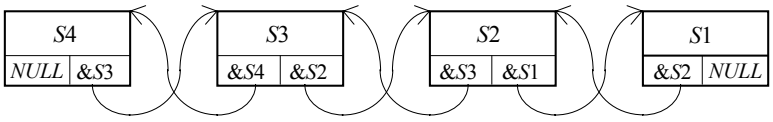
```
r1=(struct STOS*)malloc(sizeof(struct STOS));
                            // pobranie pamięci
if(!r1) goto error;         // skok po nieudanej alokacji
r1->szczyt=r;               // wskazanie poprzedniego szczytu stosu
r=r1;                       // nowy szczyt stosu
```

Likwidacja stosu

```
while(r)                    // jak długo istnieje stos, wykonuj
  { r1=r;
    r=r->szczyt;            // nowy szczyt stosu
    free(r1);               // zwolnij pamięć pobranego elementu
  }
```

7.2.3. Łańcuchy

Łańcuch ma topologię podobną do stosu, z tą różnicą, że w łańcuchu można poruszać się w dwu kierunkach. Wymaga to zadeklarowania drugiego wskaźnika, jak pokazano na rys. 7.5.



Rys. 7.5. Powiązanie struktur w łańcuch

Mając dany wskaźnik do jednego ogniwa (struktury) można obliczyć wskaźniki do wszystkich pozostałych ogniw łańcucha. Jeśli np. wskaźniki *p* oraz *p*1 zdefiniowane są jako

```
struct OGNIWO
  { . . . . . .             // opisy innych pól struktury
    struct OGNIWO *lewy, *prawy;   // wskaźniki do sąsiadów
  } *p, *p1;
```

a *p* wskazuje na ogniwo *S*2 (rys. 7.4), to wyrażenie

```
p->prawy                    // wskazuje na S1
p->lewy                     // wskazuje na S3
p->lewy->lewy               // wskazuje na S4
```

Jeśli zdefiniujemy strukturę

```
struct STOS
  { . . . . . .             // opisy innych pól struktury
    struct STOS *szczyt;    // wskaźnik do poprzedniego szczytu stosu
  } *r, *r1;
```

oraz jeśli wskaźnik *r* wskazuje na *R*3 (rys. 7.4), to można obliczyć tylko

```
r->szczyt                    // wskazanie na R2
r->szczyt->szczyt            // wskazanie na R1
```

Przykłady tworzenia i modyfikacji łańcucha

Utworzenie pierwszego ogniwa

```
p=(struct OGNIWO*)malloc(sizeof(struct OGNIWO));
                             //pobranie pamięci
if(!p) goto error;           // skok po nieudanej alokacji
p->lewy=p->prawy=NULL;       // skraje łańcucha
```

Dołączenie ogniwa do łańcucha z lewej strony (*p* wskazuje dowolne ogniwo).

```
while(p->lewy) p=p->lewy;    // p wskazuje skrajne lewe ogniwo
p1=(struct OGNIWO*)malloc(sizeof(struct OGNIWO));
                             //pobranie pamięci
if(!p1) goto error;          // skok po nieudanej alokacji
p1->prawy=p;                 // dołączenie łańcucha do ogniwa
p->lewy=p1;                  // dołączenie ogniwa do łańcucha
p1->lewy=NULL;               // lewy skraj łańcucha
```

Usunięcie skrajnego lewego ogniwa z łańcucha (*p* wskazuje dowolne ogniwo).

```
while(p->lewy) p=p->lewy;    // p wskazuje skrajne lewe ogniwo
p=p->prawy;                  // p wskazuje przedostatnie ogniwo
free(p->lewy);               // zwolnienie pamięci
p->lewy=NULL;                // lewy skraj łańcucha
```

Pytania i zadania

7.5. Uzupełnij przytoczone w rozdziale przykłady tworzenia i modyfikacji łańcucha o:
   a) dołączenie ogniwa pomiędzy *S*2 i *S*3, gdy *p* wskazuje na *S*2,
   b) usunięcie prawego skrajnego ogniwa, gdy *p* wskazuje na dowolne ogniwo,
   c) usunięcie ogniwa wskazywanego przez *p*,
   d) zwolnienie wszystkich ogniw,

7.6. Węzeł drzewa binarnego zawiera wskaźnik do tekstu oraz licznik, ile razy ten tekst wystąpił. Zaproponuj strukturę tego węzła.

7.7. Napisz instrukcje, które utworzą węzeł, zaalokują bufor na słowo i umieszczą w nim to słowo. Strukturę węzła przyjmij jak w poprzednim zadaniu.

7.8. Zaproponuj strukturę węzła drzewa do utworzenia słownika angielsko-polskiego i polsko-angielskiego. Napisz instrukcje, które utworzą węzeł dla nowego słowa.

7.9. Łańcuch pokazany schematycznie na rysunku 7.4 jest otwarty. Jakie wartości należy nadać zerowym wskaźnikom, aby zamknąć ten łańcuch w pierścień?

7.3. Struktury i funkcje

Struktura nie może być argumentem ani wynikiem funkcji. (Ograniczenia te nie dotyczą języka C++). Argumentem i wynikiem funkcji może być wskaźnik do struktury.

Przykłady funkcji obliczających długość i środek odcinka

Odcinek dany jest poprzez współrzędne swoich końców (*A*, *B*). Współrzędne punku zawiera struktura o nazwie *PUNKT*

```
struct PUNKT
  {float x, y;
  };
```

Do funkcji przekazywane są wskaźniki do punktów *A* i *B* stanowiących końce odcinka. Wynikiem funkcji *dlugosc* jest długość odcinka.

```
float dlugosc(struct PUNKT *A, struct PUNKT *B)
  { float dx, dy;
    dx=B->x-A->x;                  // przyrost współrzędnej x
    dy=B->y-A->y;                  // przyrost współrzędnej y
    return(sqrt(dx*dx+dy*dy));     // długość odcinka
  }
```

Wynikiem funkcji *srodek* jest wskazanie zaalokowanej (przez tę funkcję) struktury zawierającej współrzędne środka odcinka.

```
struct PUNKT *srodek(struct PUNKT *A, struct PUNKT *B)
  { float dx, dy, d;
    struct PUNKT *p;
    p=(struct PUNKT*)malloc(sizeof(struct PUNKT));
    if(!p) return(p);
    p->x=0.5*(B->x+A->x);          // współrzędna x środka
    p->y=0.5*(B->y+A->y);          // współrzędna y środka
    return(p);
  }
```

Funkcja wywołująca funkcję *dlugosc* powinna sprawdzić czy zwracany wskaźnik nie jest pusty. Jeśli nie jest, to po wykorzystaniu wyniku powinno się zwolnić pamięć zaalokowaną w funkcji *dlugosc*. Np.

```
struct PUNKT A, B, *s;
float d;
. . . . . .
d=dlugosc(&A, &B);
printf("Dlugosc odcinka AB wynosi %f\n", d);
s=srodek(&A, &B);
if(s)
     { printf("Srodek S(%f, %f)\n", s->x, s->y);
         free(s);
     }
. . . . . .
```

Pytania i zadania

7.10. Napisz funkcję, która: a) wyprowadzi zawartość struktury *DATA*, b) wprowadzi datę do struktury *DATA*.

7.11. Napisz funkcję, która stworzy strukturę *OSOBA* i zwróci wskaźnik do niej. Załóż, że wartości pól funkcja otrzyma: a) z klawiatury, b) przez parametry.

7.12. Napisz rekurencyjną funkcję, która znajdzie zadane słowo w drzewie, którego węzły zawierają wskaźniki do słów i liczniki wystąpień tych słów. Jeśli słowo zostanie znalezione, to zwiększa się licznik wystąpień o 1. Jeśli nie, to tworzy się dla tego słowa węzeł i dołącza się ten węzeł do drzewa, tak aby każdy węzeł w swojej lewej gałęzi zawierał tylko słowa alfabetycznie wcześniejsze, w prawej gałęzi tylko słowa późniejsze.

7.13. Napisz rekurencyjną funkcję, która zlikwiduje drzewo z poprzedniego zadania, zwalniając zarówno pamięć węzłów, jak i bufory słów.

7.4. Tablice struktur

Tablice wzajemnie powiązanych ze sobą zmiennych wygodnie jest zastąpić tablicami struktur. Przykładem może być opisana w rozdz. 2.7 tablica symboli, z których tworzone są liczby rzymskie i tablica wartości związanych z tymi symbolami. Użycie w programie dwu tablic (*char*[13][3] i *int*[13]) nie pokazuje związku pomiędzy nimi czyniąc program mniej zrozumiałym. Wygodniej jest powiązać elementy tych dwu tablic parami w struktury i stworzyć tablicę struktur. Innym przykładem może być tablica nazw państw i ich skrótów

```
struct KRAJE {
  char Symbol[4], *Nazwa;
} KR[]={{"A","Austria"},{"AUS","Australia"},{"B","Belgia"},
        {"CDN","Kanada"},{"CZ","Czechy"},{"F","Francja"},
        {"PL","Polska"},{"RUS","Rosja"},{"TJ","Chiny"}};
```

Ryzyko błędnego zainicjowania tak powiązanych tekstów jest mniejsze niż w przypadku niezależnego inicjowania dwu oddzielnych tablic.

Nie powinno się inicjować struktur automatycznych.

Pytania i zadania

7.14. Zdefiniuj i zainicjuj tablicę struktur zawierającą:
   a) nazwy miesięcy, liczbę dni w miesiącach i liczbę dni od 1 stycznia do 1 danego miesiąca,
   b) nazwy substancji, ich ciężary właściwe i temperatury topnienia,
   c) dane z tablicy Mendelejewa dla kilku pierwiastków.

   Teksty o zróżnicowanej długości powinny być umieszczane w indywidualnie alokowanych buforach. Wskaźniki do tych buforów powinny być przechowywane w strukturach.

7.5. Pola

Gdy trzeba oszczędzać pamięć, można **pakować** małe (jedno- lub wielobitowe) **obiekty w jedno słowo** definiując strukturę pól w tym słowie. Pola mogą być składowymi struktury. W definicji po nazwie pola **po dwukropku podaje się rozmiar pola** (liczbę bitów pola). Poniższa struktura definiuje zmienną *flg* zawierającą dwa jednobitowe pola: *flag*1 i *flag*2 oraz dwubitowe pole *mode*. Struktura *flg* zajmuje jedno słowo.

```
struct
  { unsigned flag1:1;          // pole jednobitowe
    unsigned flag2:1;
    unsigned mode:2;           // pole dwubitowe
  } flg;
```

Pola nie mają adresu, a ich rozmieszczenie w słowie może być różne na różnych maszynach.
Pole nie może przekraczać granicy słowa.
Pola zachowują się jak zmienne całkowite bez znaku.

Na przykład:

```
flg.flag1=flg.flag2=0;
flg.mode=2|flg.flag3;
if(flg.flag3==1) flg.flag1=1;
```

Pole bez nazwy – anonimowe – tworzy obszar nie wykorzystywanych bitów. Pole to nie jest inicjowane i nie podlega przypisaniu.
Jeśli szerokość pola ma **rozmiar 0**, to następne pole zostanie umieszczone w kolejnym słowie.

Niżej zadeklarowana struktura *SKRAJNE* zajmuje jedno słowo udostępniając w nim dwa skrajne bity.

```
struct SKRAJNE
  { unsigned  B0:1;            // bit 0 (Borland C i C++)
    unsigned :14;              // 14 bitów odstępu (bity 1–14)
    unsigned B15:1;            // bit 15
  };
```

Pytania i zadania

7.15. W implementacji Borland C++ zmienna *TextAttr* jest ośmiobitowa i zawiera atrybuty ekranu. Bity od 0 do 3 określają kolor znaku, bity od 4 do 6 – kolor tła, a bit 7 steruje migotaniem. Napisz strukturę pól zmiennej *TextAttr*.

7.16. Pewne urządzenie nadawcze ma rejestr stanu, w którym następujące bity mają znaczenie: bit 4 – SEND, bit 6 – TIE, bit 7 – TxDONE, bit 8 – RESET, bit 10 – ACTIV, bity 11 i 12 – TEST, bit 15 – ERROR. Zaproponuj odpowiednią strukturę pól zakładając, że pierwsze pole zajmuje bit 0.

7.17. Ustawienie bitu RESET zeruje nadajnik (bit ten sam się zeruje). Ustawienie bitu SEND uaktywnia (odblokowuje) logikę nadajnika. W odpowiedzi na ustawianie się bitu TxDONE należy wpisywać bajty danych do bufora, co automatycznie zeruje bit TxDONE. Opierając się na tych informacjach, napisać fragment programu obsługi nadajnika, przyjmując strukturę jego rejestrów jak w poprzednich zadaniach.

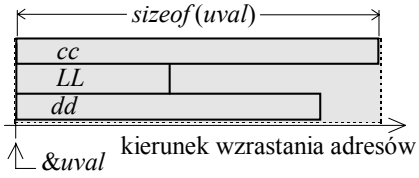
7.18. Pewien nadajnik ma rejestr buforowy, którego bity mają znaczenie: bity od 0 do 7 – dane nadawane, bit 8 – START, bit 9 – STOP, bit 10 – ABORT. Zaproponuj strukturę pól dla tego rejestru zakładając rozmieszczenie pól od bitu 0.

7.6. Unie

Unia jest zmienną, która może zawierać obiekty różnych typów. O ile w strukturze jej składowe są umieszczane szeregowo jedna po drugiej, to składowe unii są umieszczane równolegle jedna na drugiej, począwszy od tego samego adresu w pamięci.
W jednym momencie unia przechowuje wartość tylko jednej składowej – tej, której wartość została do unii wpisana najpóźniej**.**

Rozmiar unii jest równy lub większy od największego rozmiaru jej składowych. Składnia unii jest oparta na strukturach, np.

```
union u_nazwa
  { char cc[12];
    long LL;
    double dd;
  } uval;
```



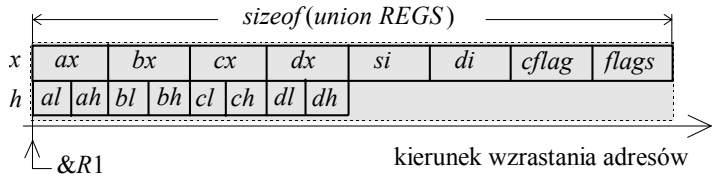
Rys. 7.6. Rozmieszczenie w pamięci składowych *cc*, *LL*, *dd*, przykładowej unii. Zmienna *uval*, jej rozmiar i wskaźnik *&uval*

Powyższa definicja definiuje zmienną *uval*, mogącą pomieścić liczby typu *long* albo *float*, albo *double*, albo 12 bajtów. Zmienna *uval* może być np. użyta do rozpakowania zmiennych rzeczywistych na bajty. Rozmiar *sizeof*(*uval*) jest równy (rys.7.6)

$$12 = max\{sizeof\,(char[12]),\ sizeof\,(long\ double)\}.$$

W pliku **dos.h** jest zdefiniowana następująca unia dwu struktur *x* oraz *h*

```
union REGS
   {struct{unsigned  ax, bx, cx, dx, si, di, cflag, flags;} x;
    struct{unsigned char al, ah, bl, bh, cl, ch, dl, dh;} h;
   } R1,R2;
```



Rys. 7.7. Organizacja pamięci unii *REGS*

Unia *REGS* posiada dwie składowe *x* oraz *h* (rys. 7.7). Obie składowe są strukturami umieszczonymi we wspólnej pamięci, począwszy od tego samego adresu. Rozmiar *sizeof* (*union REGS*) jest równy 8·*sizeof* (*unsigned*).

Unia *REGS* jest przydatna do współpracy z rejestrami mikroprocesora. Pozwala alternatywnie używać nazw rejestrów 16-bitowych i nazw rejestrów ośmiobitowych (rys. 7.7) bez wpływu na rozmieszczenie wartości tych rejestrów w pamięci. Poniższa funkcja załaduje rejestry procesora danymi zawartymi w unii wskazanej przez *inreg*, wygeneruje przerwanie o numerze 0x21 i skopiuje dane z rejestrów procesora do unii wskazanej przez *outreg*.

```
intdos(union REGS *inreg, union REGS *outregs)
```

Jeśli np. *R*1 oraz *R*2 są uniami *REGS*, to po wywołaniu

```
intdos(&R1,&R2);
```

zawartość starszego bajta akumulatora po wygenerowaniu i obsłużeniu przerwania odczytuje się jako

```
R2.h.ah
```

Jeśli zdefiniowano wskaźnik

```
union REGS pr=&R2;
```

to starszy bajt akumulatora jest dany jako

```
pr->h.ah
```

Pytania i zadania

7.19. Napisz unię, która może posłużyć do rozkładu na bajty zmiennych typu: a) *int* i *long*, b) *float* i *double*.

7.20. Używając unii wykonaj obliczenia $i1 = k \% 256$ oraz $i2 = k / 256$, które rozkładają zmienną *k* na bajty.

7.21. Funkcja *getch* zwraca kod klawisza jako liczbę typu *int* z przedziału od 0 do 255. Wynik zerowy oznacza klawisz funkcyjny, którego kod zwróci następne wywołanie funkcji *getch*. Zapisując ten kod na starszy bajt zmiennej typu *int* oraz zero na młodszy bajt tej zmiennej, można otrzymać unikalne kody typu wszystkich klawiszy w postaci pojedynczych liczb typu *int*. Zaproponuj unię do tworzenia takich kodów.

Zadania laboratoryjne

7.22. Napisz program z użyciem funkcji opracowanych do zadań 7.12 i 7.13, który stworzy uporządkowane drzewo wyrazów, a następnie wyprowadzi te wyrazy w kolejności alfabetycznej.

7.23. Program z poprzedniego zadania zmodyfikuj tak, aby pełnił on funkcję uczącego się słownika angielsko-polskiego. Program poszukuje słowa w drzewie. Jeśli słowo zostanie znalezione, jest ono wyprowadzane razem ze swoim polskim odpowiednikiem. W przeciwnym razie program prosi o podanie odpowiednika polskiego i oba słowa zapamiętuje w utworzonym nowym węźle.

7.24. Napisz program z użyciem tablicy struktur *RZYM* do konwersji arabskich liczb dziesiętnych z przedziału od 1 do 3999 na liczby rzymskie.

7.25. Uzupełnij przykładową strukturę *RZYM* o składowe, które dla danego symbolu liczby rzymskiej podają, ile razy może się on powtórzyć oraz o ile pozycji dalej jest w tablicy najbliższy symbol dozwolony po tym symbolu. Napisz program zamiany liczb rzymskich (z kontrolą ich poprawności) na arabskie liczby dziesiętne.

7.26. Napisz program, który wprowadziwszy nazwę funkcji i jej argument, znajdzie w tablicy tę nazwę oraz odpowiedni wskaźnik do funkcji związanej z tą nazwą, a następnie wykona tę funkcję i wyprowadzi wynik. Przyjmij, że elementem tablicy jest struktura zawierająca wskaźnik do tekstu i wskaźnik do funkcji.

7.27. Napisz program z użyciem unii zaproponowanej do zadania 7.21, który będzie wyprowadzać na ekran kody naciskanych klawiszy.

8. Standardowe wejście i wyjście

Prawie każdy program korzysta ze standardowych wejść lub wyjść. Wymagane prototypy funkcji, definicje makr i zmiennych zawiera plik **stdio.h**, który należy włączyć na początku programu instrukcją *#include <stdio.h>*.

8.1. Strumienie, otwieranie i zamykanie plików

Źródłem danych do programu mogą być różne urządzenia fizyczne, takie jak klawiatura monitora, pliki dyskowe czy port szeregowy. Podobnie wyniki z programu mogą być przesyłane na ekran monitora, drukarkę, do plików dyskowych lub do portu szeregowego. Wszystkie te urządzenia określa się wspólną nazwą **plik wejściowy** lub **plik wyjściowy**.

Program odbiera dane ze **strumieni wejściowych** i przekazuje dane wynikowe do **strumieni wyjściowych**. Dopiero te strumienie są kojarzone z plikami. Sposób przesyłania danych z lub do strumieni nie zależy od tego, z jakimi plikami są one skojarzone. Rozróżnia się **strumienie tekstowe**, traktowane jako zredagowany ciąg znaków i **strumienie binarne**, traktowane jako ciąg bajtów.

Utworzenie strumienia i połączenie go z plikiem nazywamy **otwarciem pliku**. Pliki otwiera funkcja *fopen*. **Zamknięcia pliku**, czyli rozłączenia pliku ze strumieniem i likwidacji strumienia, dokonuje funkcja *fclose*.

Każdy otwarty plik musi być kiedyś zamknięty.

Z chwilą uruchomienia programu są niejawnie ustanawiane trzy strumienie tekstowe:

stdin – strumień wejściowy, domyślnie skojarzony z klawiaturą monitora,
stdout – strumień wyjściowy, domyślnie skojarzony z ekranem monitora,
stderr – strumień diagnostyczny, skojarzony zawsze z ekranem monitora.

Po wykonaniu programu strumienie te są automatycznie zamykane.

Niejawnie zdefiniowane zmienne globalne *stdin, stdout* i *stderr* są wskaźnikami typu *FILE** i wskazują na struktury typu *FILE*, w których przechowywane są informacje o stanie strumieni.

Odpowiednim wywołaniem programu można skojarzyć strumienie *stdin* i *stdout* z innymi urządzeniami niż domyślne. Na przykład w systemie DOS następujące uruchomienia programu o nazwie *prog* spowodują:

prog > wyniki.txt – skojarzenie *stdout* z plikiem *wyniki.txt*,
prog > prn – skojarzenie *stdout* z drukarką,
prog < dane.1 – skojarzenie *stdin* z plikiem *dane.*1,
prog < dane > prn – skojarzenie *stdin z* plikiem *dane*, oraz *stdout* z drukarką.

Strumień *stderr* zawsze wyprowadza wyniki na ekran monitora. Strumień ten jest przeznaczony do komunikacji z operatorem – informowania o błędach oraz o stanie programu.

Do **ustanawiania strumieni** i kojarzenia ich z plikami służy funkcja *fopen* o nagłówku

```
FILE *fopen(const char *plik, const char *tryb)
```

plik – wskazuje na tekst z nazwą pliku ("con", "prn", "com" lub nazwa pliku),
tryb – wskazuje na tekst określający tryb otwarcia (do odczytu, do zapisu, itp.).

Pierwsza obowiązkowa litera określa tryb podstawowy:

r – do odczytu (otwierany plik musi istnieć),
w – do zapisu (plik jest kreowany, a jeśli istnieje, to najpierw jest kasowany),
a – do dopisywania (jeśli plik nie istnieje, jest kreowany, jeśli istnieje, to jest otwierany i przewijany na koniec).

Pozostałe opcjonalne litery modyfikują cechy trybu podstawowego i oznaczają:

t – otwarcie w trybie tekstowym,
b – otwarcie w trybie binarnym,
+ – możliwość zarówno odczytu jak i zapisu.

Jeśli żadna z liter **b** lub **t** nie występuje, to zależnie od wartości zmiennej globalnej *_fmode* plik otwiera się w trybie tekstowym, gdy ma ona wartość *O_TEXT* (wartość domyślna), lub w trybie binarnym, gdy ma czy wartość *O_BINARY*.

Jeśli plik otwarto z możliwością zapisu i odczytu, wymaga się, aby przy przejściu:

– z zapisu na odczyt opróżnić bufor strumienia wywołując funkcję *fflush*,
– z odczytu na zapis ustawić pozycję zapisu funkcją *fseek*, chyba że odczytany został koniec pliku.

W przypadku nieudanego otwarcia pliku, funkcja *fopen* daje w wyniku wskazanie puste NULL.

Zauważmy, że choć oba tryby otwarcia *r+* oraz *w+* dają zarówno możliwość odczytu, jak i zapisu, to różnią się one trybem otwarcia. Tryb *r+* podobnie jak tryb *r* wymaga, aby otwierany plik istniał. Tryb *w+* podobnie jak *w* kasuje istniejący plik i kreuje go na nowo.

Przykłady

Po zdefiniowaniu

```
FILE *fp1, *fp2, *fp3, *fp4, *fp5, *fp6, *fp7, *fp8, *fp9;
```

następujące instrukcje otwierają:

```
fp1=fopen("dane.1", "r");
```
– plik *dane.*1 do odczytu,

```
fp2=fopen("wynik.1", "w");
```
– plik *wynik.*1 do zapisu,

```
fp3=fopen("dane.b","rb");
```
– plik *dane.b* do odczytu binarnego,

```
fp4=fopen("baza.t","r+");
```
– plik *baza.t* do odczytu i zapisu (aktualizacji),

```
fp5=fopen("a:wynik", "a");
```
– plik *wynik* na dyskietce *a:* do dopisywania,

```
fp6=fopen("dane.b","rb+");
```
– plik binarny *dane.b* do aktualizacji,

```
fp7=fopen("prn", "w");
```
– drukarkę do zapisu,

```
fp8=fopen("con", "r");
```
– klawiaturę do odczytu,

```
fp9=fopen("con", "w");
```
– ekran monitora do zapisu.

Aby sprawdzić, czy pliku został otwarty prawidłowo, należy przyrównać wynik funkcji *fopen* do wskaźnika pustego np.:

```
if(fp1 == NULL) goto et1;
if((fp2=fopen("wyniki.1", "w")) == NULL) goto et2;
if(!fp3) goto et3;
```

Otwarte pliki należy zamknąć instrukcjami

```
fclose(fp1);  fclose(fp2);   . . .   , fclose(fp9);
```

Funkcja *fclose o* nagłówku

```
int fclose(FILE *stream);
```

daje w wyniku 0 lub *EOF*, gdy plik nie został zamknięty.

Równoczesnego zamknięcia pliku skojarzonego ze strumieniem *stream* i otwarcia innego pliku o podanej nazwie dokonuje funkcja

```
FILE *freopen(char *name, char *mode, FILE *stream);
```

Pytania i zadania

8.1. Z podanych wcześniej przykładów otwierania plików wypisz tryby, które:
a) otwierają pliki tylko do odczytu, b) wymagają, aby otwierane pliki istniały,
c) skracają otwierane pliki do zera, d) otwierają pliki binarne.

8.2. Wyjaśnij różnicę między wyjściowymi strumieniami *stdout* oraz *stderr*.

8.3. Napisz instrukcję, która otworzy plik w celu:
a) odczytu danych tekstowych, b) kreowania nowego pliku na wyniki,
c) dopisania kolejnej porcji wyników, d) utworzenia pliku binarnego.

8.2. Znakowe wejście i wyjście

Pojedyncze znaki i ciągi znaków wprowadza i wyprowadza się za pomocą funkcji: *fgetc, fputc, fgets* i *fputs*. Ponieważ często dane wprowadza się ze strumienia *stdin*, a wyniki wyprowadza do strumienia *stdout*, istnieją odpowiednie funkcje opracowane specjalnie dla tych strumieni. Są to funkcje: *getchar, putchar, gets* i *puts*.

Przesyłanie pojedynczych znaków realizują:

```
int fgetc(FILE *fp)              – odczyt znaku ze strumienia fp,
int getchar(void)                – odczyt znaku ze strumienia stdin,
int fputc(int c, FILE *fp)       – zapis znaku do strumienia fp,
int putchar(int c)               – zapis znaku do strumienia stdout.
```

Funkcje te dają w wyniku kod wprowadzanego (*get*) lub wyprowadzanego (*put*) znaku lub wartość EOF w przypadku wystąpienia błędu.
Parami równoważne są następujące instrukcje

```
getchar();      i     fgetc(stdin);
putchar(c);     i     fputc(c, stdin);
```

Przesyłanie ciągu znaków realizują:

```
char *fgets(char *buf,           – odczyt tekstu ze strumienia fp,
            int n, FILE *fp)
char *gets(char *buf)            – odczyt tekstu ze strumienia stdin,
int fputs(char *buf, FILE *fp)   – zapis tekstu do strumienia fp,
int puts(char *buf)              – zapis tekstu do strumienia stdout.
```

Funkcje *fgets* i *gets* dają w wyniku wskazanie *buf* bufora z wprowadzonym tekstem lub wskazanie puste NULL w przypadku błędu. Tekst wprowadzany jest do znaku nowej linii. Funkcja *fgets* wprowadza tekst ze znakiem nowej linii '\n', ale nie więcej niż $n-1$ znaków. Funkcja *gets* nie wprowadza znaku nowej linii.

Wprowadzony tekst do bufora jest w nim zakończony zerem '\0'. Funkcje *fputs* i *puts* wyprowadzają tekst z bufora *buf* (aż do ogranicznika '\0') i zwracają kod ostatniego znaku. Funkcja *puts* dopisuje na końcu znaki *cr-lf* przejścia na początek nowego wiersza.

Jeśli *fp*1 i *fp*5 wskazują strumienie z przykładów w rozdziale 8.1 oraz jeśli zmienną *c* i tablicę *buf* zdefiniowano jako:

```
int c; char buf[80];
```

to kolejne instrukcje:

```
while ((c=fgetc(fp1))!=EOF) putc(c);
while (fgets(buf, 80, fp1)!=NULL) puts(buf);
while (fgets(buf, 80, fp1)!=NULL) fputs(buf, fp5);
```

przepiszą tekst z pliku *dane.*1 do *stdout* (na ekran) znak po znaku i całymi wierszami, dopiszą tekst z pliku *dane.*1 do pliku *wyniki* na dyskietce w kieszeni *A*:.

Funkcja *fgets* pozwala ograniczyć liczbę wprowadzanych znaków i zabezpieczyć bufor przed przepełnieniem. Dlatego często stosuje się tę funkcję również do wprowadzania tekstów z klawiatury zamiast funkcji *gets*. Na przykład

```
fgets(buf, 80, stdin);
```

Pytania i zadania

8.4. Napisz program, który wyprowadzi zawartość podanego pliku tekstowego na ekran (do *stdout*).

8.5. Napisz program, który znaki wprowadzane z klawiatury aż do pierwszego znaku *Esc* prześle na drukarkę.

8.3. Formatowane wejście i wyjście

Zwykle do wprowadzania danych – zwłaszcza danych liczbowych i do wyprowadzania wyników stosuje się funkcje formatowanego wejścia i wyjścia: *scanf, fscanf, printf, fprintf*. Funkcje *fscanf* i *fprintf* mogą wymieniać dane z dowolnymi strumieniami, natomiast funkcja *scanf* pobiera dane z *stdin*, a *printf* przesyła wyniki do *stdout*.

8.3.1. Wejście

```
int fscanf(FILE *fp, char *format, parametry wzorców konwersji );
int scanf(char *format, parametry wzorców konwersji);
```

Funkcja *scanf* wprowadza dane z *stdin*. Poniższe dwie instrukcje są zatem równoważne

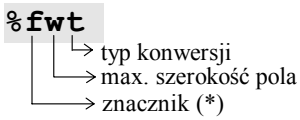
```
scanf(format, parametry wzorców konwersji);
fscanf(stdin, format, parametry wzorców konwersji);
```

Obie funkcje zwracają w wyniku liczbę wczytanych danych. Ta liczba powinna być równa liczbie parametrów wskaźnikowych odpowiadających kolejnym wzorcom konwersji w formacie. Instrukcja

```
scanf("%d%f", &n, &x);
```

wprowadza np. z *stdin* znaki dwu pól liczbowych, dokonuje konwersji pierwszego pola na liczbę typu *int* i podstawia tę liczbę pod wskazanie *&n*. Tekst drugiego pola

traktuje jak liczbę *float*, której wartość podstawia pod wskazanie &*x*. Dwie zmienne *n* oraz *x* otrzymują wczytane wartości, tak więc funkcja *scanf* zwraca wartość 2.



Rys. 8.1. Wzorzec konwersji wejściowej

Parametr *format* wskazuje na tekst, który zawiera wzorce konwersji postaci **%fwt**.

Wzorzec konwersji zaczyna się znakiem % i kończy znakiem typu konwersji. Pozostałe znaki nie są obowiązkowe. Jeśli we wzorcu występuje znacznik *, to po wprowadzeniu pola tekstowego, wynik konwersji jest pomijany. Takiemu wzorcowi nie odpowiada żaden parametr funkcji *scanf*. Liczba *w* ogranicza maksymalną liczbę znaków pola podlegających konwersji *t*. Typ konwersji musi być zgodny z typem argumentu wejściowego, jak podano w tabeli 8.1.

Tabela 8.1. Typy konwersji wejściowych

Typ konwersji	Typ argumentu	Uwagi dotyczące pola znakowego
i, *d*, *u*, *o*, *x*	*int**	postać liczby: dziesiętna, ósemkowa, heksagonalna
e, *f*, *g*	*float**	liczba w postaci dziesiętnej z wykładnikiem lub bez
s	*char*[ ]	tekst zakończony białym znakiem
[^...], [...]	*char*[ ]	tekst zakończony znakiem z zestawu [^...] lub spoza [...]
c	*char**	dowolny znak (nawet biały)
p	*void***	liczba lub para liczb heksagonalnych z dwukropkiem
n	*int**	brak pola znakowego

Do oznaczenia typu konwersji argumentów typu *long** należy użyć dużych liter *I, D, U, O, X* lub litery małe można poprzedzić kwalifikatorem l (mała litera l). Kwalifikatorów *h, l, L* używa się do oznaczenia kolejno typów *short**, *long**, *double** oraz *long double** jak podano w tabeli 8.2.

Tabela 8.2. Kwalifikatory typów konwersji wejściowych

Typ konwersji	Kwalifikator	Typ argumentu	Przykłady
i, *d*, *u*, *o*, *x*	*h*	*short**	`%hd%hx%4hx%ho`
	l	*long**	`%ld%li%5lo%4lx`
e, *f*, *g*	*l*	*double**	`%lf%le%8lf%lg`
	L	*long double**	`%Lf%Le%Lg%9Lf`

Przed zinterpretowaniem pól tekstowych pomija się białe znaki. Wyjątek stanowi konwersja %c, która wprowadza bieżący znak pliku. Konwersja %*c* wczytuje jeden znak. Ogranicznikiem pozostałych pól z wyjątkiem %[...] i %[^...] jest znak odstępu

lub dowolny biały znak. Dla konwersji [...] pole rozciąga się do napotkania dowolnego znaku spoza wymienionych w nawiasach [ ]. Dla konwersji %[^...] ogranicznikiem pola jest dowolny ze znaków wymienionych w nawiasach [^], jak pokazano w tabeli 8.3.

Tabela 8.3. Ograniczniki pól konwersji wejściowych

Typ konwersji	Ogranicznik pola
c	1 znak
[...]	znak spoza zestawu [...]
[^...]	znak z zestawu [^...]
i, *d*, *o*, *u*, *x* *e*, *f*, *g*, *s*, *p*	znak biały lub szerokość pola lub znak, który nie pasuje do pola

Konwersji %*n* nie odpowiada żadne pole znakowe. Argumentowi przypisanemu tej konwersji nadaje się wartość liczby znaków wprowadzonych podczas wykonywania funkcji *scanf*. Na przykład instrukcja

```
fscanf(stdin, "%[^\n\r]%n", TAB, &n);
```

wczyta tekst do znaku \n lub \r (a nie do spacji) do tablicy *TAB* i pod zmienną *n* podstawi liczbę wczytanych bajtów. Instrukcja

```
fscanf(stdin, "%[0123]", TAB)
```

wczyta znaki z zestawu 0, 1, 2, 3 do pierwszego innego znaku. Instrukcja

```
fscanf(stdin, "%*d%s", TAB)
```

opuści liczbę typu *int* i wczyta do tablicy *TAB* tekst do pierwszego odstępu.

Przykłady interpretacji pól

Format	Pole tekstowe	Wprowadzone wartości
`"%d%o%x"`	`14 14 14`	`14, 12, 20`
`"%i%i%i"`	`14 014 0x14`	`14, 12, 20`
`"%d%c%c%2s"`	`14 0156 0x16`	`14, ' ', '0', "15"`
`"%d%c%d"`	`123a567`	`123, 'a', 567`
`"%d%1d%6d"`	`14 015 0x16`	`14, 0, 15`
`"%2d%4f%5s"`	`123.456789`	`12, 3.45,  "6789"`
`"%2d%4f%5s"`	`123a456789`	`12, 3.0,  "a4567"`
`"%[01]%d%n"`	`10123`	`"101", 23, 5`
`"%[^45]%d"`	`21415`	`"21",  415`
`"%[^\n]%n"`	`A B C \n`	`"A B C ",  6`

Argumentami funkcji *scanf* i *fscanf* są wskaźniki. Jeśli *m* jest nazwą zmiennej typu *int*, to częstym błędem jest opuszczenie operatora adresacji **&**, co w przykładowej instrukcji **`scanf("%d", &m);`** spowoduje zapisanie wprowadzonej liczby całkowitej nie do zmiennej *m*, lecz pod przypadkowy adres.

8.3.2. Wyjście

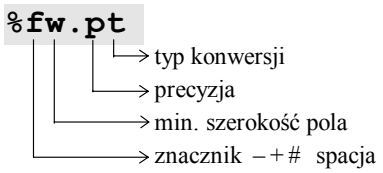
`int fprintf(FILE *fp, char *format,` *parametry wzorców konwersji*`);`
`int printf(char *format,` *parametry wzorców konwersji*`);`

Funkcja *printf* wyprowadza wyniki do *stdout*. Poniższe dwie instrukcje są zatem równoważne

`printf(format,` *parametry wzorców konwersji*`);`
`fprintf(stdout, format,` *parametry wzorców konwersji*`);`

Obie funkcje zwracają w wyniku liczbę wyprowadzonych znaków lub EOF w przypadku wystąpienia błędu.

Parametr **format** wskazuje na wzorzec wyprowadzanego tekstu. Ten wzorzec może w ustalonym tekście zawierać pola zmienne (wzorce konwersji) postaci **%fw.pt**, które są wypełniane wartościami z kolejnych argumentów.



Rys. 8.2. Wzorzec konwersji wyjściowej

Wzorzec konwersji zaczyna się znakiem % i kończy znakiem typu konwersji. Pozostałe znaki nie są obowiązkowe. Typ konwersji musi być zgodny z typem wyprowadzanego argumentu jak ujęto w tabeli 8.4.

Tabela 8.4. Typy konwersji wyjściowych

Typ konwersji	Typ argumentu	Uwagi dotyczące pola znakowego
i, *d*, *u*, *o*, *x*, *X*	*int*	postać liczby: dziesiętna, ósemkowa, heksagonalna
e, *f*, *g*, *E*, *G*	*float*	liczba w postaci dziesiętnej z wykładnikiem lub bez
s	*char*[ ]	tekst
c	*char*	dowolny znak
p	*void**	liczba lub para liczb heksagonalnych z dwukropkiem

Użycie dużych liter *X, E, G* do oznaczenia typu konwersji spowoduje użycie dużych liter do wyprowadzenia liczb szesnastkowych i wykładnika *E* liczby rzeczywistej.

Do konwersji argumentów typu *short*, *long*, *double*, *long double* należy poprzedzić typ konwersji kwalifikatorem *h, l* lub *L*, tak jak dla funkcji wejściowych.

Szerokość *w* określa minimalną liczbę znaków pola wyjściowego. Jeśli wynik wymaga pola szerszego, to zostanie on wyprowadzony w polu szerszym, w przeciwnym razie pole wyniku zostaje uzupełnione (najczęściej spacjami) do *w* znaków. Znaki uzupełnienia są dodawane po tekście wyniku lub przed nim, gdy użyto znacznika – (minus). Jeśli szerokość *w* wyrażono liczbą zaczynającą się od cyfry 0, to napis zostanie uzupełniony zerami z lewej strony do szerokości *w*.

Precyzja *p* określa dla konwersji typu *e, f* lub *g* liczbę cyfr (domyślnie 6) po kropce dziesiętnej, a dla konwersji typu *s* – maksymalną do wyprowadzenia liczbę znaków tekstu.

Znacznikiem mogą być wyszczególnione w tabeli 8.5 znaki: –, +, # lub spacja.

Tabela 8.5. Znaczniki wzorca konwersji wyjściowych

Znacznik	Wpływ znacznika na wyjściowe pole
– (minus)	justowanie lewostronne (uzupełniające spacje dopisywane za tekstem)
+ (plus)	wyprowadzana liczba jest poprzedzona znakiem + lub –
spacja	wyprowadzana liczba jest poprzedzona znakiem spacji lub –
#	liczba ósemkowa jest poprzedzona zerem a heksagonalna 0*x* lub 0*X*

Szerokość *w* i precyzja *p* mogą być zastąpione znakiem *. Wartość *w* lub *p* jest brana wtedy jako wartość kolejnych argumentów funkcji *fprintf* lub *printf*. Te argumenty muszą być typu *int*. Na przykład poniższe dwie instrukcje są równoważne

```
fprintf(stdout, "%*.*lf", 6, 2, y);
fprintf(stdout, "6.2lf", y);
```

Przykłady interpretacji formatów

Format	Argumenty	Pole wyjściowe
`":%5.3s=%-5.3s="`	`"Borland", "Borland"`	`:######Bor=Bor######=`
`"=%d=%x=%o="`	`29, 29, 29`	`=29=1d=35=`
`"=%#X=%#o="`	`29, 29`	`=0x1D=035=`
`"=%+d=% d="`	`57, 57`	`=+57=###57=`
`"=%4d=%-4d=%04d="`	`57, 57, 57`	`=######57=57######=0057=`
`"=%c=%3c="`	`68, 68`	`=D=######D=`
`"=%lf=%.1lf="`	`68, 68`	`=68.000000=68.0=`
`"=%2d=%2lf="`	`1234, 1.234`	`=1234=1.234000=`

`":%.1E=%*.*lf:"`	`-1289, 6, 1, 1.51`	`:-1.3E+003=######### 1.5:`

Przykład

Przepisanie liczb rzeczywistych z pliku „*DANE*" do pliku „*WYNIK*" w postaci tabelarycznej

```
int i, n, w, p;
float x;
FILE *fp1, *fp2;

printf("Podaj: liczbe kolumn, szer. kolumny, precyzje ");
scanf("%d%d%d", &n, &w, &p);

fp1=fopen("DANE", "r");
if(!fp1)
    { fprintf(stderr,"\nBrak pliku DANE\n");
      exit(1);
    }
if((fp2=fopen("WYNIK","w"))==NULL)
    { fprintf(stderr,"\nNie mozna otworzyć pliku WYNIK!\n");
      exit(2);
    }

for(i=0; fscanf(fp1,"%f", &x)==1; i++)
  {if(i%n==0)fprintf(fp2,"\n|");  // kreska po nowej linii
   fprintf(fp2,"%*.*f |",w,p,x);  // liczba, spacja i kreska
  }

fclose(fp2);
fclose(fp1);
```

8.3.3. Formatowane przekształcanie pamięci

Formatowane przekształcenia pamięci (czytanie z bufora i zapis do bufora) realizują funkcje:

```
int sscanf(char *b, char *format, parametry wzorców konwersji);
int sprintf(char *b, char *format, parametry wzorców konwersji);
```

Posługiwanie się tymi funkcjami jest takie samo jak funkcjami *fscanf* i *fprintf*, z tą różnicą, że pierwszy argument funkcji *sscanf* i *sprintf* nie wskazuje strumienia, lecz bufor tekstowy (typu *char**) w pamięci. Bufor funkcji *sprintf* musi mieć taki rozmiar,

aby mógł pomieścić cały wyprowadzany tekst. Funkcja *sprintf* wyprowadza tekst zakończony ogranicznikiem '\0'.

8.3.4. Wejście i wyjście na konsolę

Wejście bezpośrednio z klawiatury i wyjście na ekran realizują funkcje:

```
int cscanf(char *format, parametry wzorców konwersji);
int cprintf(char *format, parametry wzorców konwersji);
```

Funkcje te są w swej obsłudze bardzo zbliżone do funkcji *scanf* i *printf*. Należy jednak pamiętać, że znak '*n*' w formacie funkcji *cprintf* spowoduje tylko przejście do nowej linii bez ustawienia kursora na początek tej linii. Przesunięcie na początek linii uzyskuje się znakiem '*r*'. Stąd konieczność używania obu tych znaków jednocześnie. Jeśli w buforze klawiatury znajdują się oba znaki (*cr-lf*), to funkcja *cscanf* będzie analizować te znaki oddzielnie.

Funkcje *cscanf* i *cprintf* nie są funkcjami standardowego wejścia i wyjścia. Ich prototypy są umieszczone w pliku *conio.h*. Funkcja *cscanf* czyta bezpośrednio z bufora klawiatury. Funkcja *cprintf* wyprowadza tekst wraz z aktualnie obowiązującymi atrybutami bezpośrednio na ekran do aktualnego okna.

Pytania i zadania

8.6. Napisz instrukcję wywołania funkcji *scanf*, która wczyta wartości zmiennych:
a) *n*, *m*, *L*, b) *n*, *a*, *b*, *x*, c) *L*, *W*, *x*, *a*,
jeśli wcześniej zdefiniowano*: int n, m*; *long L*; *float a,b*; *double x*; *long double W*;

8.7. Napisz instrukcję funkcji *scanf*, która do tablicy zdefiniowanej jako *char tx*[80]; wczyta tekst:
a) złożony z jednego wyrazu, b) złożony z wielu wyrazów.
Zagwarantuj że tablica *tx* nie zostanie przepełniona.

8.8. Zbiór skojarzony ze strumieniem *fp*1 zawiera tekst "12.34567abc". Do jakiego typu zmiennych i jakie wartości zostaną wprowadzone formatem
a) `"%f%s"` b) `"%d%f%c"` c) `"%d%c%ld"`
e) `"5x%2f%3d%s"` f) `"%o%4f%4d%2s"` g) `"%2f%*3s%"`

8.9. Napisać format do wczytania
a) daty w postaci trzech liczb oddzielonych myślnikami lub kropkami (*dd-mm-rrrr*),
b) trzech liczb całkowitych dodatnich, wplecionych w dowolny tekst.

8.10. Napisz format drukowania trzech liczb kolejno typu *short*, *short* oraz *int*, tworzących datę w postaci *dd-mm-rrrr*, na przykład 25-04-1998.

8.11. Tablicę N liczb całkowitych (typu *int*) wyprowadź w słupku, podpisując jednostki pod jednostkami, dziesiątki pod dziesiątkami itd.

8.12. Instrukcję drukowania w zad. 8.11 zmień tak, aby powstały trzy słupki z postacią dziesiętną, ósemkową i heksagonalną tej liczby. Niech liczby ósemkowe będą poprzedzone zerem, a heksagonalne napisem "0*x*".

8.13. Wyprowadź jednym formatem dwie liczby typu *double* oraz ich sumę z dokładnością 0.001 w postaci np.: $1.500+2.750=4.250$ lub np.: $1.500-2.750=-1.250$, gdy drugi składnik jest ujemny.

8.14. Liczby a, b, c typu *float* są współczynnikami wielomianu $y=ax^2+bx+c$. Napisz ten wielomian jedną instrukcją, wypisując współczynniki z zadaną dokładnością n cyfr po kropce dziesiętnej, gdzie n jest zmienną typu *int*.

8.4. Binarne wejście i wyjście

Gdy dane i wyniki są w zrozumiałej postaci znakowej, stosuje się formatowane wejście i wyjście. Gdy jednak dane są przygotowane przez program i przeznaczone do przetworzenia przez ten sam lub inny program, korzystniej jest je przechowywać w postaci binarnej, czyli takiej, jaką mają one w pamięci komputera.

Wprowadzanie i wyprowadzanie binarne nie wymaga stosowania konwersji do/z postaci znakowej. Jest więc ono wykonywane prościej i szybciej.

Binarne wejście i wyjście realizują funkcje:

```
size_t fread(void *buf, size_t size, size_t n, FILE *fp);
size_t fwrite(void *buf, size_t size, size_t n, FILE *fp);
```

Typ *size_t* jest w implementacji Borland C++ zdefiniowany jako *unsigned*.

Obie te funkcje przesyłają n bloków o rozmiarze po *size* bajtów. W rezultacie między strumieniem *fp* a buforem *buf* jest przesyłanych $n \cdot size$ bajtów. Po poprawnym wykonaniu funkcje te zwracają liczbę n, a w przeciwnym razie liczbę mniejszą od n.

Na przykład, poniższe funkcje wczytują ze strumienia *fp*3 12 elementów (dowolnego typu) tablicy *A*, a następnie zapisują te elementy z tablicy *A* do strumienia *fp*6. Obie funkcje powinny zwrócić liczbę 12.

```
fread(A, sizeof(A[0]), 12, fp3);
fwrite(A, sizeof(A[0]), 12, fp6);
```

Pytania i zadania

8.15. Napisz program, który będzie wczytywać liczby typu *float* z klawiatury i będzie te liczby wyprowadzać do pliku binarnego.

8.16. Program z poprzedniego zadania zmodyfikuj tak, aby pozwalał on w zależności od życzenia dopisywać dane do istniejących w pliku lub wyprowadzać te dane do pustego pliku.

8.5. Sterowanie buforem i pozycją pliku

Do sterowania pozycją pliku służą kolejne cztery funkcje, natomiast funkcja *fflush* opróżnia bufor pliku.

```
int fseek(FILE *fp, long offset, int baza);
long ftell(FILE *fp);
int fgetpos(FILE *fp, fpos_t *pos);
int fsetpos(FILE *fp, fpos_t *pos);
int fflush(FILE *fp);
```

Każdy otwarty plik znajduje się w określonej pozycji, od której będzie on zapisywany lub odczytywany kolejną instrukcją wejściową lub wyjściową. Funkcje *ftell* oraz *fgetpos* określają tę pozycję. Funkcje *fseek* i *fsetpos* ustawiają plik w zadanej pozycji. Parametr *baza* funkcji *fsee*k może być równy 0, 1 lub 2 i określa, czy plik ustawić w pozycji *offset* bajtów od początku, od pozycji aktualnej czy od końca.

Przykłady

Instrukcja	Ustawiana pozycja pliku

`fseek(fp, 0L, 0);`	początek pliku
`fseek(fp, 20L, 0);`	20 bajtów za początkiem pliku
`fseek(fp, 24L, 1);`	24 bajty za pozycją aktualną
`fseek(fp, -24L, 1);`	24 bajty przed pozycją aktualną
`fseek(fp, -20L, 2);`	20 bajtów przed końcem pliku
`fseek(fp, 0L, 2);`	koniec pliku

Aby zminimalizować liczbę operacji między strumieniem a plikiem, przesyłane dane wejściowe i wyjściowe są buforowane we wspólnym buforze. Wyprowadzane dane są najpierw gromadzone w buforze. Rzeczywisty (fizyczny) zapis do pliku następuje samoczynnie po zapełnieniu się bufora oraz w chwili zamknięcia pliku. Istnieją sytuacje, w których należy opróżnić bufor, wymuszając fizyczne przesłanie danych do pliku. Gdy np. po zapisie (*fprintf*, *fwrite*) następuje odczyt (*fscanf*, *fread*), zawartość bufora jest wypełniana odczytanymi danymi. Dane nie przekazane fizycznie do pliku wyjściowego zostałyby zniszczone. Innym przykładem jest zatrzymanie wydruku w celu zmiany kartki papieru na drukarce. W tym momencie cały tekst przeznaczony do wydrukowania na zdejmowanej stronie musi być fizycznie wydrukowany, a nie czekać w buforze.

Funkcja *fflush* opróżnia bufor, usuwając z niego dane wejściowe lub wyprowadzając fizycznie dane wyjściowe do pliku.

Przykład korekty danych w pliku fp6

```
fpos_t poz;                          // long poz;
. . . . . .
fflush(fp6);                         // opróżnienie bufora (gdy był zapis)
poz=ftell(fp6);                      // zapamiętanie pozycji
fread(A, sizeof(*A), 10, fp6);
. . . . . .
fseek(fp, poz, 0);                   // odtworzenie pozycji
fwrite(A, sizeof(*A), 10, fp6);
. . . . . .
```

Najczęstsze błędy to:

- Użycie niewłaściwego kodu konwersji, np. %*d* lub %*f* w miejsce %*ld* lub %*lf*.
- Użycie zmiennych zamiast wskaźników w wywołaniu funkcji *scanf*.
- Brak funkcji *fflush* przy przejściu z zapisu na odczyt lub przed wstrzymaniem programu.

- Zapominanie o znaku przejścia na początek wiersza '\r' przy znaku '\n' w formacie funkcji *cprintf*.
- Pozostawianie otwartych plików.

Złym nawykiem jest używanie wyjścia *stdout* zamiast *stderr* do komunikacji z operatorem. Wyniki z programu są przemieszane z komunikatami, a w przypadku skierowania strumienia wyjściowego nie na ekran zrywa się kontakt programu z operatorem.

Pytania i zadania

8.17. Plik binarny zawiera liczby typu *double*. Napisać program, który:
a) wyznaczy ile liczb jest w pliku bez czytania tych liczb,
b) wczyta *n*-tą w kolejności liczbę, wypisze tę liczbę na ekranie, a w pliku na jej miejsce wpisze zero.

8.18. Plik binarny zawiera liczby typu *float*. Wszystkie dodatnie liczby tego pliku zastąpić ich pierwiastkami, nie tworząc nowego pliku.

Zadania laboratoryjne

8.19. Napisać program, który w zależności od życzenia wypisze na drukarce lub ekranie tablice funkcji trygonometrycznych w postaci tabeli o kolumnach x (w stopniach i minutach), $\sin(x)$, $\cos(x)$, $\mathrm{tg}(x)$, zatrzymując obliczenia po każdych 64 wierszach w celu zmiany papieru. Przedział zmienności x, liczba punktów i dokładność wyników powinny być zadawane. Program powinien akceptować nazwę pliku podaną w parametrach wywołania programu.

8.20. Napisać dwa programy. Pierwszy obliczy tablice funkcji trygonometrycznych jak w poprzednim zadaniu i wyprowadzi wartości x (w radianach), $\sin(x)$, $\cos(x)$ i $\mathrm{tg}(x)$ do pliku binarnego. Drugi program odczyta te wartości i wyprowadzi w postaci tabelarycznej na ekran.

8.21. Napisać program, który po uruchomieniu napisze, który raz został uruchomiony. Wskazówka: do pliku wynikowego *.*exe* dopisać na koniec dwa zerowe bajty za pomocą specjalnie napisanego programu. W programie właściwym wykorzystać te dwa bajty. Nazwę swojego pliku wraz z pełną ścieżką w każdym programie wskazuje argument *argv*[0] funkcji *main*.

9. Tryb tekstowy

W trybie tekstowym podstawowym elementem ekranu jest znak. Domniemanym oknem tekstowym jest cały ekran. Użycie funkcji ekranowych wymaga dołączenia do programu pliku **conio.h**. Wykonanie każdego programu zaczyna się w trybie tekstowym.

9.1. Ekran i jego atrybuty

Ekran tekstowy jest podzielony na kolumny i wiersze. Liczba kolumn ekranu jest jego szerokością wyrażoną w znakach. Liczba wierszy jest wysokością ekranu. Wymiary ekranu (*screenwidth, screenheight*) są zależne od trybu tekstowego i mogą wynosić (40, 25), (80, 25) lub (80, 50). Do zmiany trybu tekstowego służy funkcja

```
void textmode(int Mode);
```

gdzie *Mode* = { BW40, C40, BW80, C80, MONO, C4350, LASTMODE }.

W trybie *Mode* = BW40 lub C40 ekran ma 40 kolumn. W trybach: BW80, C80, MONO i C4350 ekran ma 80 kolumn. W trybie C4350 ekran ma 50 wierszy, natomiast w pozostałych trybach – 25 wierszy. Parametr *Mode* = LASTMODE ustanawia tryb obowiązujący w chwili uruchomienia programu.

Na przykład instrukcja

```
textmode(C80);
```

ustanawia tryb tekstowy kolorowy o 80 kolumnach i 25 wierszach.

Położenie znaku na ekranie podaje się w układzie współrzędnych (*x, y*), gdzie *x* jest numerem kolumny, a *y* jest numerem wiersza. Wiersze i kolumny numeruje się od 1. Lewy górny róg ekranu ma współrzędne (1, 1).

Wyprowadzany na ekran znak jest umieszczany w miejscu wyróżnionym **kursorem**. W obszarze ekranu można zdefiniować **okno tekstowe** za pomocą funkcji

```
void window(int x1, int y1, int x2, int y2);
```

gdzie ($x1$, $y1$), ($x2$, $y2$) są ekranowymi współrzędnymi odpowiednio lewego górnego i prawego dolnego rogu okna. Domyślnym oknem jest cały ekran. Na przykład instrukcja

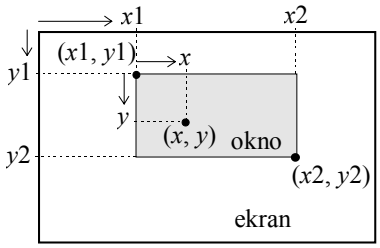
```
window(30, 10, 40, 15);
```

ustanawia okno o 11 kolumnach i 6 wierszach. W zależności od trybu, okno to znajduje się na lewo od środka lub przy prawym skraju ekranu. Najmniejsze okno zawiera jeden znak. Na przykład

```
window(30, 12, 30, 12);
```

ustanawia okno złożone ze znaku w 30. kolumnie i 12. wierszu ekranu.

Wszystkie operacje tekstowe na ekranie dotyczą bieżącego okna, które ma wszystkie właściwości ekranu (np. zawijanie tekstu czy przewijanie). Współrzędne kursora podaje się zawsze względem lewego górnego rogu okna, jak pokazano na rys. 9.1.



Rys. 9.1. Współrzędne okna w ekranie i znaku w oknie

W bieżącym oknie wykonywane są następujące funkcje:

```
void clrscr(void)              – czyszczenie całego okna,
void clreol(void)              – czyszczenie do końca linii (wiersza),
void delline(void)             – usunięcie linii (wiersza) w oknie,
void insline(void)             – wstawienie linii (wiersza) w oknie,
int wherex(void)               – określenie współrzędnej x kursora,
int wherey(void)               – określenie współrzędnej y kursora,
void gotoxy(int x, int y)      – przesunięcie kursora do punktu (x, y).
```

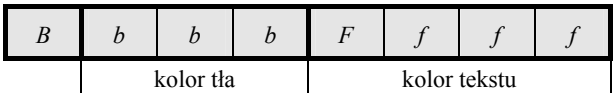
Tak więc np. funkcja *clrscr*( ) wyczyści tylko bieżące okno tekstowe, nie zaś cały ekran. Po ustanowieniu okna jak na rys. 9.1 instrukcjami

```
window(x1, y1, x2, y2);
gotoxy(x, y);
```

kursor znajdzie się w kolumnie nr $x1+x-1$ oraz w wierszu nr $y1+y-1$ względem lewego górnego rogu ekranu. Dla np. $x1=30$, $y1=10$, $x=4$, $y=2$ będzie to kolumna nr 33 i wiersz nr 11. Funkcje *wherex*( ); i *wherey*( ); określą położenie kursora w oknie (tu 4 i 2).

Każdy znak na ekranie jest w pamięci reprezentowany przez dwa bajty: kod i atrybut. Kod określa znak, natomiast atrybut określa sposób wyświetlania (kolory, migotanie, rozjaśnienie). Aktualnie obowiązujący atrybut jest przechowywany w zmiennej *TextAttr*. Wyprowadzany na ekran znak jest umieszczany w pamięci ekranu razem z atrybutem pamiętanym w zmiennej *TextAttr*.

Najstarszy bit atrybutu – bit *B* (rys. 9.2) włącza migotanie (gdy $B=1$), kolejne trzy bity *bbb* oznaczają numer koloru tła, a najmłodsze cztery bity *Ffff* oznaczają numer koloru tekstu. Bit *F* nazywany jest bitem jasności – ustawiony oznacza jaśniejsze kolory tekstu.



Rys. 9.2. Struktura bajta atrybutu

Zmiennej globalnej *TextAttr* wartość swojego argumentu nadaje funkcja

```
void textattr(int attr)
```

Na przykład poniższe instrukcje ustawią:

```
textattr(0x1E);   – żółte znaki na niebieskim tle,
textattr(0x9E);   – migające żółte znaki na niebieskim tle,
textattr(0);      – niewidoczne czarne znaki na czarnym tle.
```

Poniższy program utworzy w trybie C80 na czarnym ekranie dwa rzędy po cztery prostokąty w możliwych kolorach tła.

```
#include <conio.h>                    // włączenie opisu grafiki tekstowej
void main(void)
{int i, x1, y1;
 textattr(WHITE);                     // białe znaki na czarnym tle
 clrscr();                            // wyczyszczenie ekranu
 for(i=0; i<8; i++)
   { textattr(i<<4);                  // ustawienie koloru tła nr i
     x1=10+15*(i&3);                  // obliczenie współrzędnych okna
     y1=5+(i&4)<<1;
     window(x1, y1, x1+12, y1+6);     // ustanowienie okna
     clrscr();                        // wyświetlenie tła w oknie
   }
}
```

Poszczególne pola zmiennej *TextAttr* można zapisywać oddzielnie funkcjami

`void textbackground(int kolor)` – ustawienie koloru tła (*bbb*),
`void textcolor(int kolor)` – ustawienie koloru tekstu (*Ffff*),
`void highvideo(void)` – ustawienie bitu jasności ($F = 1$),
`void lowvideo(void)` – wyzerowanie bitu jasności ($F = 0$).

W pliku *conio.h* zdefiniowano zmienne wyliczeniowe z nazwami kolorów.

Tabela 9.1. Zdefiniowane nazwy kolorów

Wartość liczbowa i nazwa		Kolor	Uwagi
0	BLACK	czarny	ciemne kolory tekstu i kolory tła (w bajcie atrybutu na kolor tła przeznaczono trzy bity)
1	BLUE	niebieski	
2	GREEN	zielony	
3	CYAN	turkusowy	
4	RED	czerwony	
5	MAGENTA	karmazynowy	
6	BROWN	brązowy	
7	LIGHTGRAY	jasnoszary	
8	DARKGRAY	ciemnoszary	jasne kolory tekstu
9	LIGHTBLUE	jasnoniebieski	
10	LIGHTGREEN	jasnozielony	
11	LIGHTCYAN	jasnoturkusowy	
12	LIGHTRED	jasnoczerwony	
13	LIGHTMAGENTA	jasnokarmazynowy	
14	YELLOW	żółty	
15	WHITE	biały	
128	BLINK	migotanie	bit *B*

Posługiwanie się tymi nazwami, nie zaś numerami czyni program bardziej czytelnym. Na przykład:

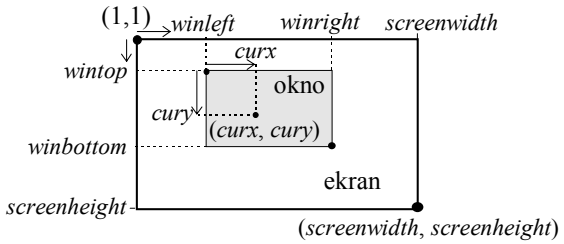
`textcolor(YELLOW);` – ustawia żółte znaki,
`textcolor(WHITE|BLINK);` – ustawia białe migające znaki,
`textbackground(RED);` – ustawia czerwone tło,
`lowvideo();` – ściemnia kolor znaku (np. zmieni kolor żółty na brązowy).

Informacje o stanie systemu tekstowego do wskazanej struktury wpisuje funkcja

```
void gettextinfo(struct text_info *Info);
```

gdzie struktura *struct text_info* jest zdefiniowana następująco:

```
struct text_info{
    unsigned char winleft, wintop,        // (x1,y1) – lewy górny róg okna
                  winright, winbottom,    // (x2,y2) – prawy dolny róg
                  attribute,              // wartość zmiennej TextAttr
                  normattr,               // początkowa wartość Textattr
                  currmode,               // numer trybu tekstowego Mode
                  screenheight,           // wymiary ekranu
                  screenwidth,
                  curx, cury;             // położenie kursora w oknie
                 } ;
```



Rys. 9.3. Interpretacja składowych struktury *struct text_info*

Na przykład

```
struct text_info ti;                  – definicja struktury ti
int x1, y1;                           – współrzędne lewego górnego rogu
gettextinfo(&ti);                     – pobranie informacji o ekranie
x1=ti.screenwidth>>2;                 – x1= szerokość ekranu / 4
y1=ti.screenhight>>2;                 – y1= wysokość ekranu / 4
window(x1, y1, 3*x1, 3*y1);           – utworzenie okna
textbackground(GREEN);                – tło zielone
textcolor(YELLOW);                    – znaki żółte
clrscr();                             – zapisanie tła w oknie
```

Powyższe instrukcje niezależnie od trybu tekstowego utworzą w środku ekranu zielony prostokąt o wymiarach dwa razy mniejszych niż ten ekran. Ewentualnie wyprowadzany tekst będzie koloru żółtego.

Pytania i zadania

9.1. Wyznacz ekranowe położenie kursora, jeśli w trybie C80 przed instrukcją *gotoxy*(12, 5); wystąpiła instrukcja
a) *window*(1, 1, 80, 25); b) *window*(10, 5, 30, 15);
c) *window*(60, 8, 80, 25); d) *window*(8, 20, 28, 25);
Wykonaj odpowiednie rysunki.

9.2. Wyznacz wartości zmiennej *TextAttr* jeśli kolory tła i tekstu wynoszą kolejno:
a) BLACK i WHITE b) BLACK i WHITE + BLINK
c) BLUE i YELLOW d) BROWN i GREEN + BLINK
e) LIGHTGRAY i BLACK f) RED i LIGHTMAGENTA

9.3. Napisz funkcję, o nagłówku *void scr_invers*(*int inv*), która dla *inv*###0 ustawi tło w kolorze znaku, a znak w kolorze tła, natomiast dla *inv*=0 ustawi kolory obowiązujące w chwili uruchomienia programu.

9.2. Wejście i wyjście na konsolę

Poza funkcjami obsługi strumieni *stdin* i *stdout*, które z reguły są skojarzone z klawiaturą i z ekranem, istnieją funkcje bezpośrednio obsługujące klawiaturę i ekran. Są to funkcje:

```
int kbhit (void)                  – testowanie bufora klawiatury,
int getch(void)                   – pobranie znaku z bufora,
char *cgets(char *buf)            – wprowadzenie tekstu z konsoli,
int putch(int, ch)                – wyprowadzenie znaku na ekran,
int cscanf(char *format, ...)     – wprowadzenie zredagowane,
int cprintf(char *format, ...)    – wyprowadzenie zredagowane.
```

Funkcja *kbhit*( ) zwraca wartość różną od zera, gdy w buforze klawiatury znajduje się nie odczytany znak. Funkcja ta nie pobiera znaku z bufora. Znak można pobrać funkcją *getch*( ), która daje w wyniku kod tego znaku. Funkcja ta nie daje echa na ekranie. Na przykład instrukcja

```
while (kbhit()) getch();          – opróżnia bufor klawiatury.
```

Jeśli bufor klawiatury jest pusty, to *getch*( ) czeka na naciśnięcie klawisza. Niektóre klawisze, takie jak np. strzałki i inne klawisze specjalne, wprowadzają do bufora po dwa bajty, z których pierwszy jest zerowy. W takim przypadku należy funkcję *getch*( )

wywołać powtórnie. Na przykład poniższe dwie funkcje zwracają indywidualne kody wszystkich klawiszy

```
int scr_getc1(void)
  { int c=getch();
    if(c) return(c);
    return(((c=getch())<128) ? c+128 : c);
  }

int scr_getc2(void)
  { int c=getch();
    if(c) return(c);
    return(getch()<<4);
  }
```

Funkcja *cgets*(*buf*) wprowadza z klawiatury do bufora *buf* ciąg znaków do znaku nowej linii, nie dłuższy jednak niż liczba znaków zapisana w *buf* [0]. Wprowadzony tekst jest zakończony znakiem zerowym i nie zawiera końcowego znaku nowej linii. Pierwszy znak testu jest wprowadzany do *buf* [2]. Funkcja *cgets*(*buf*) nadaje bajtowi *buf* [1] wartość liczby rzeczywiście wprowadzonych znaków. Funkcja *cgets*(*buf*) zwraca wskazanie do *buf* [2]. Na przykład, aby nie przepełnić zdefiniowanego bufora, można funkcję *cgets* wywołać następująco:

```
char tx[40];
tx[0]=38;       // ograniczenie tekstu do 37 znaków + '\0'
cgets(tx);      // wprowadzenie tekstu
```

Parametry funkcji *csanf* są takie jak funkcji *scanf*, z tym że *cscanf* wprowadza dane bezpośrednio z bufora klawiatury (bez pośrednictwa strumienia *stdin*). Funkcje *cgets* i *csanf* mogą zostawić w buforze znak przejścia do nowej linii, który będzie pobrany w kolejnym wywołaniu tych funkcji lub funkcji *getch*( ).

Funkcji wyjściowych *putch*, *cputs* i *cprintf* używa się podobnie jak funkcji *putchar*, *puts* i *printf* . Zasadniczą różnicą w działaniu tych funkcji jest to, że *puts* i *printf* wyprowadzają jedynie kody znaków i nie respektują aktualnie ustawionego okna. Funkcje ekranowe *putch*, *cputs* i *cprintf* umieszczają na ekranie w oknie znaki wraz z aktualnym atrybutem *TextAttr*.

Funkcje ekranowe nie wstawiają dodatkowych znaków przejścia do nowej linii lub przejścia na początek linii. Tak więc np. znaki przejścia do nowej linii '*n*' w funkcji *printf* powinny być w *cprintf* uzupełnione znakiem powrotu na początek wiersza '*r*'. Na przykład do wyprowadzenia dwóch wartości typu *double* w dwóch wierszach należy użyć instrukcji:

```
cprintf("\r\nX=%lf\r\nY=%lf\r\n", x, y);
```

Pytania i zadania

9.4. Napisz funkcję, która zbada naciśnięty klawisz *T* lub *N* w odpowiedzi typu *Tak*/*Nie*. Zignoruj klawisze naciśnięte wcześniej oraz naciskane inne klawisze z wyjątkiem klawisza *Esc*, który należy zinterpretować jako *N*.

9.5. Tablica tekstowa *tx* zawiera tekst "\5\0\0...\0". Jaka będzie zawartość tej tablicy, jeśli w odpowiedzi na instrukcję *cgets*(*tx*); wprowadzono tekst „Borland"?

9.6. Jak będzie wyglądać tekst, który wyprowadzono instrukcją:

a) `cputs("\r\nJezyk C\nKompilator\n Borland C++\r\n");`
b) `cputs("\r\nKompilator\n\r\nBorland C++\r\n");`
c) `cprintf("\r\nK=%d\n\nN=%d\r\nJ=%d\r\nL=%d ",5,6,7,9);`

9.3. Odtwarzanie ekranu

Do zapamiętania i odtworzenia części ekranu służą funkcje:

```
int gettext(int x1, int y1, int x2, int y2, void *buf)
int puttext(int x1, int y1, int x2, int y2, void *buf)
```

Parametry $x1$, $y1$, $x2$, $y2$ są ekranowymi współrzędnymi ($x1$, $y1$) i ($x2$, $y2$) wierzchołków zapamiętywanego lub odtwarzanego prostokąta, podobnie jak przyjęto w funkcji *window* (rys. 9.1).

Funkcja *gettext*($x1$, $y1$, $x2$, $y2$, *buf*) przepisuje kolejno kody i atrybuty znaków wskazanego prostokąta wierszami do bufora *buf*. Tak więc bufor ten powinien móc pomieścić co najmniej dwa razy tyle bajtów, co jest znaków we wskazanym prostokącie, czyli co najmniej $2(x2-x1+1)(y2-y1+1)$ bajtów.

Funkcja *puttext*($x1$, $y1$, $x2$, $y2$, *buf*) przenosi kolejne bajty z bufora *buf* do pamięci ekranu, która odpowiada prostokątowi wskazanemu przez punkty ($x1$, $y1$) i ($x2$, $y2$). W przypadku pomyślnego wykonania obie funkcje zwracają wartość różną od zera.

Przykłady

`char buf[4000];`
`gettext(1, 1, 80, 25, buf);`
`puttext(1, 1, 80, 25, buf);`
	definicja bufora dla całego ekranu
	zapamiętanie ekranu w *buf*
	odtworzenie ekranu z *buf*

`gettext(1, 1, 40, 12, buf);`
`puttext(41, 1, 80, 12, buf);`
`puttext(1, 13, 40, 24, buf);`
`puttext(41, 13, 80, 24, buf);`
	zapamiętanie pierwszej ćwiartki ekranu i zapisanie jej w trzech pozostałych ćwiartkach

`gettext(x1, y1, x2, y2, buf);`
`n=2*(x2-x1+1)*(y2-y1+1);`
`for(i=1; i<n; i+=2)`
`buf[i]|=0x80;`
`puttext(x1, y1, x2, y2, buf);`
	spowodowanie migotania w zadanym prostokącie (zapamiętanie w buforze, zmiana atrybutów, odtworzenie z migotaniem)

`gettext(x1, y1, x2, y2, buf);`
`n=2*(x2-x1+1)*(y2-y1+1);`
`kolor=BLUE<<4;`
`for(i=1; i<n; i+=2)`
`{buf[i]&=0x8F;`
`buf[i]|=kolor;}`
`puttext(x1, y1, x2, y2, buf);`
	zmiana tła na niebieskie w zadanym prostokącie (zapamiętanie w buforze, zmiana koloru tła, odtworzenie z nowym tłem)

Pytania i zadania

9.7. Ile co najmniej bajtów powinien mieć bufor użyty w instrukcji *gettext*(*x*1, *y*1, *x*2, *y*2, *buf*);, jeśli punkty (*x*1, *y*1) i (*x*2, *y*2) wynoszą:
a) (6, 3) i (23, 7), b) (30, 10) i (40, 20), c) (*x*, *y*) i (*x*+15, *y*+5).

9.8. Napisz funkcję, która w zadanym prostokącie:
a) ustawi tło w kolorze tekstu, a tekst w kolorze tła,
b) ustawi zadane kolory tła i tekstu.

Zadania laboratoryjne

9.9. Napisz program, który podany fragment ekranu będzie przesuwał w kierunku wskazywanym przez strzałkę naciskanego klawisza. Odsłaniane fragmenty ekranu powinny być odtwarzane.

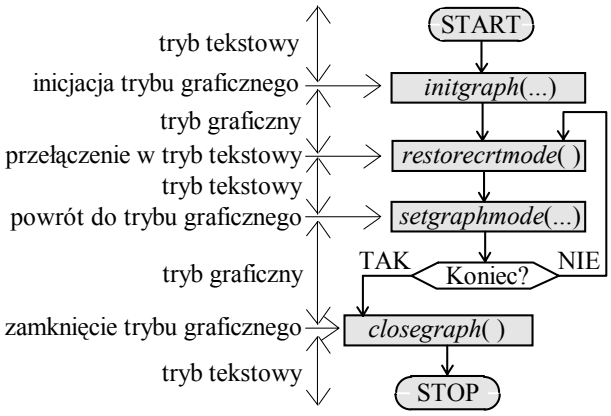
9.10. Napisz program, który będzie wyprowadzać na ekran w kolumnach liczby pierwsze. Upakuj jak najwięcej kolumn na ekranie. Do pozycjonowania kolumn użyj funkcji *window*.

9.11. Napisz funkcję menu, która wypisze na ekranie listę tekstów, podświetli wybrany strzałkami tekst, a po naciśnięciu klawisza *Enter* wywoła funkcję związaną z wybranym tekstem. Po naciśnięciu klawisza *Esc* lista tekstów menu znika z ekranu, ekran jest odtwarzany do postaci sprzed wywołania funkcji menu i następuje wyjście z tej funkcji.

9.12. Zmodyfikuj funkcję menu z poprzedniego zadania, tak aby wybierana klawiszem *Enter* funkcja mogła być też funkcją menu.

10. Tryb graficzny

W trybie graficznym podstawowym elementem ekranu jest piksel, podobnie jak w trybie tekstowym znak. Piksel jest punktem na ekranie i jest jednolicie wypełniony kolorem o numerze od 0 do *getmaxcolor*( ). Piksele (podobnie jak znaki) są ułożone w kolumnach i wierszach, a numer kolumny *x* oraz numer wiersza *y* są współrzędnymi (*x, y*) piksela. Punkty na ekranie mogą mieć współrzędne w granicach od (0, 0) do (*getmaxx*( ), *getmaxy*( )). Liczba kolumn *getmaxx*( ), i wierszy *getmaxy*( ) zależy od karty graficznej komputera i trybu graficznego. Użycie trybu graficznego wymaga więc rozpoznania karty graficznej oraz inicjacji trybu graficznego, to znaczy załadowania odpowiedniego sterownika (drivera) dla tej karty. Do programu należy włączyć plik nagłówkowy **graphics.h**. Linker powinien przeglądać bibliotekę graficzną, co należy włączyć w opcjach *Options* | *Linker* | *Libraries*.



Rys. 10.1. Przełączanie trybów tekst–grafika

Inicjacja trybu graficznego jest czynnością czasochłonną. Jeśli program musi wielokrotnie używać na przemian trybu graficznego i tekstowego, to po zainicjowaniu

grafiki funkcją *initgraph* należy przełączać ekran do trybu tekstowego funkcją *restorecrtmode* i powracać do trybu graficznego funkcją *setgraphmode*.

Funkcji *closegraph*, która zamyka grafikę powinno się użyć dopiero wtedy, gdy tryb graficzny nie będzie już więcej potrzebny. Schemat postępowania ilustruje rys.10.1.

10.1. Otwieranie i zamykanie grafiki

Uniwersalne programy powinny same rozpoznawać kartę graficzną. Rozpoznania tego dokonuje funkcja:

```
void detectgraph(int *Driver, int *Mode)
```

Pod wskazane zmienne podstawia ona dwie wartości: numer karty graficznej (*Driver*) oraz maksymalny (o maksymalnej rozdzielczości) dla tej karty numer trybu (*Mode*). Tryb graficzny inicjuje funkcja

```
void initgraph(int *Driver, int *Mode, char *Path)
```

Jeśli wskazywana zmienna *driver=DETECT*, to funkcja *initgraph* automatycznie wywołuje funkcję *detectgraph* i inicjuje tryb na rozpoznanej karcie z maksymalną jej rozdzielczością. Na przyk³ad

```
int driver=DETECT, mode;
initgraph(&driver, &mode, "C:\\BC3\\BGI\\");
```

Zmienna *driver* otrzymuje wartość numeru rozpoznanej karty graficznej, a zmienna *mode* maksymalny numer trybu dozwolony dla tej karty.

Można też inicjować grafikę na określonej karcie w zadanym trybie nadając przed wywołaniem *initgraph* zmiennym *driver* i *mode* numer karty graficznej i numer trybu. Można tu posłużyć się zdefiniowanymi zmiennymi wyliczeniowymi: 1-CGA, 2-MCGA, 3-EGA, 4-EGA64, 5-EGAMONO, 6-IBM8514, 7-HERCMONO, 8-ATT400, 9-VGA, 10-PC3270. Na przyk³ad

```
int driver=VGA, mode=0;
initgraph(&driver, &mode, "C:\\BC3\\BGI\\");
```

Ostatni parametr *Path* wskazuje tekst ze ścieżką do sterownika (*drivera*) karty graficznej. Driver karty graficznej jest poszukiwany w kartotece podanej ścieżką *Path*, a następnie w kartotece bieżącej.

Otwarcie grafiki może skończyć się niepowodzeniem. Dlatego należy sprawdzić stan systemu graficznego za pomocą funkcji

```
int graphresult(void)
```

zwracającej kod błędu ostatniej operacji graficznej. Po pozytywnym wykonaniu się operacji, funkcja ta zwraca liczbę: **grOk**.

Do **przełączania trybów** służą funkcje

`void restorecrtmode(void)` – z trybu graficznego do trybu tekstowego,
`void setgraphmode(int Mode)` – z trybu tekstowego do trybu graficznego.

Zamknięcia trybu graficznego dokonuje funkcja

`void closegraph(void)`

Zwalnia ona pamięć przydzieloną sterownikowi karty graficznej (*driver*), pamięć zajmowaną przez krój pisma i przywraca tryb tekstowy.

Poniższy program rozpozna kartę graficzną i wykreśli linię na przekątnej ekranu.

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

void main()
  {
    int driver=DETECT, mode, e;
    initgraph(&driver, &mode, "\\BC3\\BGI\\");
    if((e=graphresult())!=grOk)      // zapamiętanie numeru błędu
      {
        cprintf("\r\nNie mozna otworzyc grafiki\r\n");
        cprintf(grapherrormsg(e)); // informacja o błędzie nr. e
        exit(1);
      }

    lineto(getmaxx(), getmaxy());
    getch();                         // zatrzymanie ekranu graficznego
    closegraph();
  }
```

Pytania i zadania

10.1. Dlaczego nie zaleca się otwierać grafiki w programie więcej niż jeden raz?

10.2. Jeśli dwie kolejne operacje graficzne są wywołaniem funkcji *graphresult*( ), to dlaczego za drugim razem funkcja ta zawsze zwróci wartość *grOk*?

10.2. Sterowanie ekranem i kursorem

Operacje na ekranie lub w oknie

```
void cleardevice(void)                          – czyszczenie ekranu,
void clearviewport(void)                        – czyszczenie okna,
void setviewport(int x1, int y1,                – utworzenie okna.
          int x2, int y2, int Clip)
```

gdzie ($x1$, $y1$), ($x2$, $y2$) określają położenie okna a $Clip = 1$ powoduje, że wykresy wykraczające poza okno będą obcinane – lub gdy $Clip = 0$ – nie będą.

Parametry okna można odczytać funkcją

```
void getviewsettings(struct viewporttype *v)
```

gdzie struktura *struct viewporttype* jest zdefiniowana jako

```
struct viewporttype {
       int left, top,        // (x1, y1) – lewy górny róg okna
       int right, bottom,    // (x2, y2) – prawy dolny róg okna
       int clip; };          // sposób obcinania
```

Operacje na kursorze

```
int getmaxx(void)             – poziomy rozmiar ekranu w pikselach,
int getmaxy(void)             – pionowy rozmiar ekranu w pikselach,
int getx(void)                – pozioma współrzędna kursora w oknie,
int gety(void)                – pionowa współrzędna kursora w oknie,
void moveto(int x, int y)     – przesunięcie kursora do punktu (x, y),
void moverel(int x, int y)    – przesunięcie kursora o wektor (x, y).
```

Zapamiętanie rysunku

```
void getimage(int x1,int y1,int x2,int y2,void *buf)
                                                – zapamiętanie,
void putimage(int x1,int y1,void *buf,int Op)   – odtworzenie,
unsigned imagesize(int x1,int y1,int x2,int y2) – rozmiar bufora.
```

Parametry ($x1$, $y1$), ($x2$, $y2$) określają lewy górny i prawy dolny róg zapamiętywanego prostokąta. Do odtworzenia podaje się tylko położenie lewego górnego wierzchołka. Parametr *Op* określa sposób nakładania się odtwarzanego obrazu z obrazem istniejącym na ekranie. Zależnie od wartości

Op = COPY_PUT, XOR_PUT, OR_PUT, AND_PUT, NOT_PUT,
odtwarzany obraz będzie przykrywać istniejący lub będzie łączyć się z nim według funkcji XOR, OR, AND lub NOT.

Wybór strony (dotyczy tylko kart EGA, VGA i Herkules)

`void setactivepage(int Page)` – wybór strony aktywnej,
`void setvisualpage(int Page)` – wybór strony wyświetlanej.

Jeśli nie można wybrać drugiej strony, należy zmniejszyć numer trybu graficznego.

Pytania i zadania

10.3. Napisz instrukcje, które lewą górną ćwiartkę ekranu powielą w pozostałych ćwiartkach.

10.4. Napisz instrukcje, które: a) zapamiętają prostokątny fragment ekranu i utworzą w nim okno graficzne, b) odtworzą poprzedni stan ekranu (zapamiętany fragment, poprzednie okno graficzne, położenie kursora).

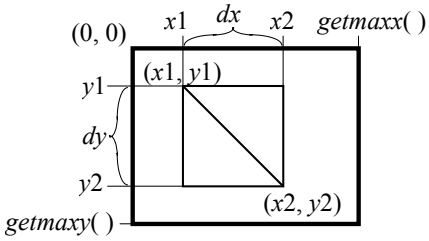
10.3. Wykreślanie linii i figur

Punkty i ich kolory

`void putpixel(int x, int y, int Color)`
– nadanie punktowi (*x*, *y*) koloru *Color*,
`int getmaxcolor(void)` – numery kolorów od 0 do *getmaxcolor*( ),
`void setbkcolor(int Color)` – kolor tła,
`void setcolor(int Color)` – kolor punktów i linii.

Linie proste – odcinki

`void line(int x1, int y1, int x2, int y2)`
– odcinek bez przesunięcia kursora,
`void linerel(int dx, int dy)` – odcinek względem kursora,
`void lineto(int x, int y)` – odcinek od kursora do (*x, y*),
`void drawpoly(int N, int *tab)` – linia łamana,
`void rectangle(int x1, int y1, int x2, int y2)` – prostokąt.



Rys. 10.2. Wykreślanie odcinków

Prostokąt z przekątną jak na rys. 10.2 można wykreślić kilkoma metodami:

```
rectangle(x1, y1, x2, y2);
moveto(x1, y1);
lineto(x2, y2);
```

albo

```
rectangle(x1, y1, x2, y2);
line(x1, y1, x2, y2);
```

albo

```
int T[]={x1,y1,x2,y1,x2,y2,
         x1,y2,x1,y1,x2,y2};
drawpolly(5, T);
```

Linie krzywe – łuki

```
void circle(int x, int y, int R)                          – okrąg,
void arc(int x, int y, int Alfa, int Beta, int R)         – łuk,
void ellipse(int x, int y, int Alfa, int Beta,            – elipsa.
             int Rx, int Ry)
```

Powyższe funkcje kreślą łuki o środku w (*x, y*) od kąta *Alfa* do *Beta* (w stopniach względem osi *X*). Rodzaj linii i jej grubość definiuje się funkcją *setlinestyle*. Sposób nakładania się linii definiuje się funkcją *setwritemode*.

```
void setlinestyle(int Styl, unsigned Wzor, int Grubosc)
void setwritemode(int Mode)
```

gdzie parametry *Styl* i *Grubosc* określają styl kreślonej linii (por. tabela 10.1).

Tabela 10.1. Parametry stylu linii

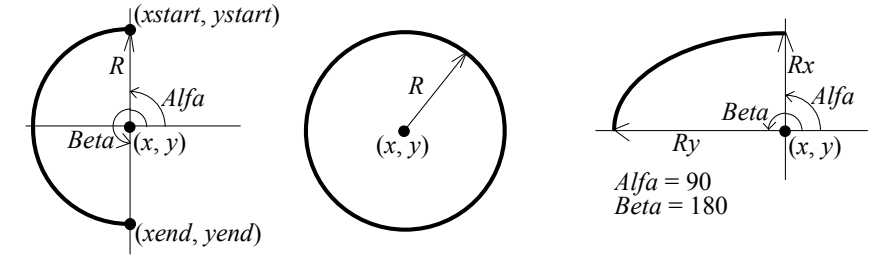
Styl:	SOLID_LINE	– ciągła	**Grubość:**	NORM_WIDTH	– cienka
	DOTTED_LINE	– kropkowa		THICK_WIDTH	– gruba
	CENTER_LINE	– centrowana			
	DASHED_LINE	– przerywana	**Mode:**	COPY_PUT	– nakładanie
	USERBIT_LINE	– wg *Wzor*		XOR_PUT	– odwracanie

Współrzędne łuku wykreślonego funkcją *arc* można określić za pomocą funkcji *getarccoords*.

```
void getarccoords(struct arccoordstype *A);
```

Struktura *struct arccoordstype* jest zdefiniowana jak niżej.

```
struct arccoordstype
 { int x, y;           // współrzędne środka
 int xstart, ystart;   // współrzędne początku łuku
 int xend, yend;       // współrzędne końca łuku
 };
```



arc(*x*, *y*, *Alfa*, *Beta*, *R*); circle(*x*, *y*, *R*); ellipse(*x*, *y*, *Alfa*, *Beta*, *Rx*, *Ry*);

Rys. 10.3. Wykreślanie łuków

Wypełnione figury

```
void bar(int x1, int y1, int x2, int y2)          – prostokąt,
void bar3d(int x1, int y1, int x2, int y2,        – słupek,
                           int Depth, int Flg)
void fillpoly(int N, int *tab)                    – wielokąt,
void fillellipse(int x, int y, int Rx, int Ry)    – elipsa,
void pieslice(int x, int y,                       – wycinek koła,
                    int Alfa, int Beta, int R)
void sector(int x, int y, int Alfa, int Beta,     – sektor elipsy.
                           int Rx, int Ry)
```

Obszar z punktem (*x, y*) ograniczony linią o numerze *Border* można wypełnić za pomocą funkcji

```
void floodfill(int x, int y, int Border);
```

Sposób i kolor wypełniania figur można określić za pomocą jednej z dwu funkcji

```
void setfillstyle(int Pattern, int Color);
void setfillpattern(char *Wzor, int Color);
```

Dla drugiej funkcji należy zdefiniować w ośmiobajtowej tablicy *Wzor* własny wzór 8×8 punktów. Pierwszą funkcją można wybrać wzór standardowy, nadając zmiennej *Pattern* wartości podane w tabeli 10.2.

Tabela 10.2. Style wypełnienia

EMPTY_FILL	– brak wypełnienia	LTBKSLASH_FILL	– linie ukośne
SOLID_FILL	– ciągły kolor *Color*	HATCH_FILL	– siatka pionowa
LINE_FILL	– linie poziome grube	XHATCH_FILL	– siatka ukośna
LTSLASH_FILL	– linie pochyłe	INTERLEAVE_FIL L	– linie splecione
SLASH_FILL	– linie pochyłe grube	WIDE_DOT_FILL	– kropki
BKSLASH_FILL	– linie ukośne grube	CLOSE_DOT_FILL	– kropki gęste

Pytania i zadania

10.5. Napisz instrukcje, które wykreślą w pobliżu środka ekranu:
   a) otwartą kopertę,
   b) szachownicę 8×8 pól,
   c) dwa współśrodkowe okręgi i elipsę styczną do nich,
   d) wycinek koła z cięciwą (cięciwę linią przerywaną).

10.4. Sterowanie kolorami

Kolory są ponumerowane od 0 do *getmaxcolor*( ). W przypadku niektórych kart graficznych (np. VGA) poszczególnym numerom można przypisać kolory używając funkcji *setpalette* lub funkcji *setallpatette*.

```
int getmaxcolor(void)          – maksymalny numer koloru,
void setbkcolor(int Color)     – ustawienie koloru tła,
void setcolor(int Color)       – ustawienie koloru punktów i linii,
int getbkcolor(void)           – odczytanie koloru tła,
int getcolor(void)             – odczytanie koloru punktów i linii,
void setpalette(int Nr, int Color)  – zmiana koloru w palecie,
void setallpalette(struct palettetype *p)  – ustawienie całej palety,
void getpalette(struct palettetype *p)  – odczytanie palety kolorów,
unsigned getpixel(int x, int y)        – odczytanie koloru punktu,
```

gdzie struktura *struct palettetype* jest zdefiniowana jako

```
struct palettetype { unsigned char size;
                     signed char colors[16];};
```

Identyfikatory kolorów są tu takie same jak dla trybu graficznego. Na przykład instrukcje

```
setpalette(BLUE, RED);
setpalette(BLUE, BLACK);
```

spowodują, że kolor niebieski (kolor nr 1) zostanie zastąpiony kolorem czerwonym, a następnie kolorem czarnym. Zmiana ta natychmiast będzie widoczna na ekranie.

Stosowanie liczbowych numerów kolorów zamiast zmiennych wyliczeniowych z nazwami kolorów jest złym nawykiem, ponieważ czyni program mniej czytelnym.

Pytania i zadania

10.6. Napisz instrukcje, które zmienią kolor punktów na kolor tła, a kolor tła na kolor punktów.

10.7. Napisz własną wersję funkcji *setallpalette* używając funkcji *setpalette*.

10.8. Wykorzystując to, że zmiana koloru w palecie jest widoczna natychmiast na ekranie, napisz instrukcje, które najpierw uczynią rysunek niewidocznym, a potem przywrócą mu poprzedni wygląd.

10.5. Wykreślanie tekstów

W trybie graficznym teksty są wykreślane podobnie jak wszystkie inne obiekty. Teksty mogą być bitowe i kreskowe. Domyślnym rodzajem jest tekst bitowy, którego znaki są prostokątami o wymiarze 8×8 punktów. Teksty kreskowe zapewniają dobrą jakość wizualną niezależnie od powiększenia, ale ich krój musi być ładowany z pliku dyskowego podobnie jak *driver* karty graficznej. Kroje czcionek są podobnie jak driver poszukiwane w kartotece określonej parametrem *Path* w funkcji *initgraph* lub w kartotece aktualnej. W każdej chwili w pamięci może znajdować się co najwyżej jeden automatycznie załadowany krój kreskowy. Więcej krojów można załadować funkcją

```
int registerbgifont(void(*Fontbgi)(void))
```

Odpowiednie fonty muszą być przetworzone programem BINOBJ.EXE z postaci *.CHR do postaci *.OBJ (np. w celu dołączenia kroju *trip.chr* komendą *binobj trip* wykonaną w podkartotece BGI kompilatora). Przetworzone kroje muszą być dołączone do programu na etapie linkowania.

Teksty mogą być wykreślane poziomo od lewej do prawej i pionowo z dołu do góry. Krój czcionki, kierunek wykreślania i powiększenie tekstu określa się funkcją

```
void settextstyle(int Font, int Kierunek, int Rozmiar)
```

Parametry: *Fontbgi, Font, Kierunek* i *Rozmiar* mogą przybierać wartości podane w tabeli 10.3.

Tabela 10.3. Fonty graficzne

Fontbgi	Font	Kierunek
	0 – DEFAULT_FONT	HORIZ_DIR – poziomy VERT_DIR – pionowy
triplex_font	1 – TRIPLEX_FONT	
small_font	2 – SMALL_FONT	
sansserif_font	3 – SANS_SERIF_FONT	
gothic_font	4 – GOTHIC_FONT	
script_font	5 – SCRIPT_FONT	**Rozmiar**
simplex_font	6 – SIMPLEX_FONT	krotność powiększenia
triplex_scr_font	7 – TRIPLEX_SCR_FONT	lub
complex_font	8 – COMLEX_FONT	USER_CHAR_SIZE
european_font	9 – EUROPEAN_FONT	
bold_font	10– BOLD_FONT	

Jeśli *Rozmiar* = USER_CHAR_SIZE, to teksty kreskowe mogą być rozciągane w poziomie i w pionie. Wcześniej należy ustawić współczynniki rozciągania. Poniższa funkcja ustawia poziomy współczynnik równy *xMul* / *xDiv* oraz pionowy współczynnik równy *yMul* / *yDiv* .

```
void setusercharsize(int xMul, int xDiv, int yMul, int yDiv);
```

Teksty wykreśla się funkcjami

```
void outtext(char *text)
```
– tekst względem pozycji kursora,

```
void outtextxy(int x,int y,char *text)
```
– tekst względem pozycji (*x, y*).

Na przykład, poniższe instrukcje wykreślą pionowo powiększony pięć razy tekst „*TRIPLEX*", a następnie wykreślą poziomo tekst „*GOTHIC*" rozciągnięty 7/3 razy w poziomie i 5/2 razy w pionie.

```
settextstyle(TRIPLEX_FONT, VERT_DIR, 5);
outtext("TRIPLEX");
setusercharsize(7, 3, 5, 2);
settextstyle(GOTHIC_FONT, HORIZ_DIR, USER_CHAR_SIZE);
outtext("GOTHIC");
```

Aby wypisać wartości zmiennych, należy najpierw przetworzyć te wartości na tekst (np. funkcjami: *itoa, ltoa, ultoa*) lub wyprowadzić sformatowany tekst do bufora za pomocą funkcji *sprintf*. Na przykład jeśli *buf* i *x* zdefiniowano jako *char buf* [80]; *float x*;, to poniższe instrukcje wykreślą wartość zmiennej *x*

```
sprintf(buf, "X=%5.2f", x);
outtext(buf);
```

Po zdefiniowaniu fontu i rozmiaru, poniższe funkcje wyznaczają pionowy i poziomy rozmiar tekstu w pikselach

`int textheight(char *text)` – pionowy rozmiar tekstu,
`int textwidth(char *text)` – poziomy rozmiar tekstu.

Na przykład rozmiar pionowy *ny* i poziomy *nx* tekstu *Borland C++* obliczą instrukcje:

```
ny=textheight("B");
nx=textwidth("Borland C++");
```

Tekst jest wykreślany względem bieżącej pozycji kursora. Jeżeli jest on wykreślany poziomo, pozycja kursora przesuwa się na koniec tekstu. Może on być również wykreślany względem pozycji (*x, y*) bez przesuwania kursora. Usytuowanie tekstu względem pozycji kursora lub (*x, y*) definiuje się funkcją

```
void settextjustify(int Horiz, int Vert)
```

W zależności od wartości parametrów *Horiz* i *Vert* tekst jest umieszczany tak, aby pozycja kursora lub (*x, y*) znajdowała się, jak podano w tabeli 10.4

Tabela 10.4. Parametry justowania tekstu

Horiz		Vert	
LEFT_TEXT	– z lewej strony tekstu	BOTTOM_TEXT	– na dole tekstu
CENTER_TEXT	– w środku tekstu	CENTER_TEXT	– w środku tekstu
RIGHT_TEXT	– z prawej strony tekstu	TOP_TEXT	– u góry tekstu

Na przykład, poniższe instrukcje umieszczą tekst *Borland* tak, że punkt (*x, y*) znajdzie się pod literą *l* niezależnie od tego, czy tekst będzie kreślony w pionie czy w poziomie

```
settextjustify(CENTER_TEXT, BOTTOM_TEXT);
outtextxy(x, y, "Borland");
```

Pytania i zadania

10.9. Wykreśl tekst „Borland C++” wybranym fontem kreskowym w środku ekranu, tak aby wysokość tekstu zajmowała 12% wysokości ekranu, a szerokość tekstu 80% szerokości ekranu.

10.10. Na ekranie są narysowane osie układu współrzędnych ze środkiem w środku ekranu. Opisz oś pionową z odstępem *dy*, jeśli zakres ekranu pokrywa wartości od y_{min} do y_{max}
a) $y_{min} = -4.2$, $y_{max} = 4.2$, $dy = 1$,
b) $y_{min} = -1.7$, $y_{max} = 1.7$, $dy = 0.5$.

10.6. Skalowanie

Aby rysunek miał na ekranie właściwe proporcje, matematyczne współrzędne punktów tego rysunku muszą być przeliczone na współrzędne ekranowe. Uzależniając to przeliczenie od wymiarów ekranu (*getmaxx*( ), *getmaxy*( )), zapewnimy jednakowy kształt rysunku niezależnie od karty graficznej.

Niech zmienne ciągłe *X, Y* przyjmują wartości od $X_{\min}$ do $X_{\max}$ oraz od $Y_{\min}$ do $Y_{\max}$. Niech punkty $(X_{\min}, Y_{\min})$ i $(X_{\max}, Y_{\max})$ odpowiadają kolejno lewemu górnemu i prawemu dolnemu rogowi ekranu. Ekranowe współrzędne (*x, y*) dowolnego punktu (*X, Y*) oblicza się ze wzoru

$$x = a_x\left(X - X_{\min}\right), \qquad y = a_y\left(Y_{\max} - Y\right),$$

gdzie

$$a_x = \frac{getmaxx(\,)}{X_{\max} - X_{\min}}, \qquad a_y = \frac{getmaxy(\,)}{Y_{\max} - Y_{\min}}.$$

Ekranowe współrzędne (*x, y*) mogą być użyte w funkcjach graficznych opisanych w dalszych rozdziałach.

Jeżeli rysunek (wykres) ma być wykonany w pewnym prostokącie, to skalowanie należy wykonać względem rozmiarów prostokąta, a współrzędne ekranowe (*x, y*) należy przesunąć o współrzędne lewego górnego rogu prostokąta.

Jeżeli wymiary ekranowe prostokąta oznaczy się przez *xm* i *ym* oraz położenie lewego górnego rogu prostokąta w punkcie (*x*1, *y*1), odpowiednie wzory na współczynniki skalujące oraz przeliczenia współrzędnych matematycznych na współrzędne ekranowe będą następujące:

$$a_x = \frac{xm}{X_{\max} - X_{\min}}, \qquad a_y = \frac{ym}{Y_{\max} - Y_{\min}},$$

$$x = x1 + a_x\left(X - X_{\min}\right), \qquad y = y1 + a_y\left(Y_{\max} - Y\right).$$

Odpowiedni program wykreślający dwa okresy funkcji $\sin(X)$ w prostokącie o wymiarach dwa razy mniejszych od wymiarów ekranu i położonym w środku ekranu pokazano poniżej.

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

```
main()
{ int driver=DETECT, mode, err;
  char *path="C:\\BC3\\BGI\\";
  int x1, y1, xm, ym, x, y;
  double ax, ay, X,
        Xmin=0, Xmax=12.6, dX=0.2, Ymin=-1, Ymax=1;
  initgraph(&driver, &mode, path);
  if((err=graphresult())!=grOk)
    { cprintf(grapherrormsg(err));
      return(1);
    }
  xm=getmaxx()/2;                          // rozmiary prostokąta
  ym=getmaxy()/2;
  x1=getmaxx()/4;                          // lewy górny róg prostokąta
  y1=getmaxy()/4;
  ax=xm/(Xmax-Xmin);                       // współczynniki skalujące
  ay=ym/(Ymax-Ymin);
  rectangle(x1,y1,x1+xm,y1+ym);            // kreślenie prostokąta
  moveto(x1, y1+ay*(Ymax-sin(Xmin)));      // ustawienie kursora
  for(X=Xmin+dX; X<Xmax; X+=dX)
    { x=x1+ax*(X-Xmin);                    // ekranowe współrzędne
      y=y1+ay*(Ymax-sin(X));
      lineto(x, y);                        // kreślenie odcinka do (x, y)
    }

  getch();                                 // zatrzymanie ekranu
  closegraph();
  return(0);
}
```

Jeżeli stosuje się ustalone współrzędne ekranowe lub ustalone współczynniki skalujące w miejsce współrzędnych i współczynników zależnych od rozmiarów ekranu *getmaxx*( ) i *getmaxy*( ),wykreślane rysunki wyglądają różnie na różnych typach kart graficznych.

Pytania i zadania

10.11. Podaj współrzędne środka ekranu oraz współrzędne wierzchołków prostokąta o bokach cztery razy mniejszych niż boki ekranu i położonego w środku prawej dolnej ćwiartki ekranu.

10.12. Jak przeliczyć współrzędne matematyczne (X, Y) na współrzędne ekranowe (x, y), aby sporządzić wykres funkcji $Y = \sin(X) + \cos(5X)$ dla X zmieniającego się od 0 do 2### oraz aby wykres wypełnił:
a) cały ekran, b) górną połowę ekranu,
c) dolną połowę ekranu, d) prawą górną ćwiartkę ekranu.

10.7. Monitorowanie systemu graficznego

Następujące funkcje zwracają wskaźnik do nazwy sterownika karty graficznej, nazwy trybu graficznego, numer i maksymalny numer trybu graficznego oraz maksymalny numer koloru:

`char *getdrivername(void)` – nazwa sterownika karty graficznej,
`char *getmodename(void)` – nazwa trybu graficznego,
`int getgraphmode(void)` – numer trybu graficznego,
`int getmaxmode(void)` – maksymalny numer trybu graficznego,
`int getmaxcolor(void)` – maksymalny numer koloru.

Kolejne funkcje wypełniają wskazane struktury informacjami o aktualnym stylu kreślenia linii i wypełniania obszarów zamkniętych, wykreślania tekstu i położenia okna:

```
void getlinesettings(struct linesettingstype *Info);
void getfillsettings(struct fillsettingstype *Info);
void gettextsettings(struct textsettingstype *Info);
void getviewsettings(struct viewporttype *Info);
```

Użytkownik może na ośmiu bajtach podać własny wzór wypełniania za pomocą funkcji

```
void getfillpattern(char *Wzor);
```

Definicje struktur użytych w powyższych funkcjach są następujące

```
struct linesettingstype {
   int linestyle; unsigned upattern; int thickness;};

struct fillsettingstype {int pattern; int color; };

struct textsettingstype {
   int font, direction, charsize, horiz, vert;};

struct viewporttype {int left, top, right, bottom, clip; };
```

Składowe powyższych struktur odpowiadają parametrom funkcji: *setlinestyle*, *setfillstyle*, *settextstyle* i *settextjustify* oraz *setviewport*. Wskaźnik *Wzor* wskazuje na ośmiobajtową tablicę podobnie jak w funkcji *setfilpattern*.

Funkcja monitorowania błędu o numerze zwracanym przez funkcję *graghresult*( ).

```
char *grapherrormsg(int Nr)
```

Wynikiem jest wskazanie tekstu związanego z błędem numer *Nr*. Przykład użycia tej funkcji pokazano w programie z rozdziale 10.1.

Pytania i zadania

10.13. Napisz instrukcje, które w trybie graficznym wykreślą tekst z nazwą sterownika karty graficznej i nazwą trybu graficznego.

10.14. Napisz instrukcje, które w trybie graficznym wykreślą tekst objaśniający błąd operacji graficznej.

Zadania laboratoryjne

10.15. Napisz program, który zainicjuje tryb graficzny z rozpoznaniem karty graficznej i wykreśli pierścień o zewnętrznym promieniu równym 40% wysokości ekranu i wewnętrznym promieniu równym 10% tej wysokości. Wypróbuj wypełnianie pierścienia różnymi wzorami.

10.16. Napisz program, który wykreśli na ekranie układ współrzędnych (*X, Y*), opisze ten układ, a następnie dla sukcesywnie podawanych wartości *a, fi* będzie wypisywać te wartości i wykreślać dla nich wykresy funkcji $x=\cos(t)$, $y=\sin(a\,t+fi)$ dla *t* zmieniającego się od 0 do 2###. Każdy wykres powinien pojawiać się zaraz po podaniu dla niego danych, a poprzedni wykres powinien znikać.

10.17. Napisz program, który zadany tekst wypisze na ekranie wszystkimi możliwymi fontami kreskowymi, jeden pod drugim poziomo, a następnie jeden obok drugiego pionowo. Zmodyfikuj program tak, aby teksty zajęły cały ekran.

10.18. Napisz program, który rozpozna kartę graficzną i wypisze nazwę sterownika tej karty, liczbę jej trybów graficznych oraz dla każdego trybu wypisze jego nazwę, maksymalne współrzędne i liczbę kolorów.

11. Przetwarzanie tekstów

11.1. Konwersja liter i rozpoznanie znaków

Są to funkcje (lub makrodefinicje) zamiany małych liter na duże i odwrotnie oraz funkcje sprawdzające, czy znak o danym kodzie należy do określonej grupy znaków:

`int toupper(int c)` – zamiana małych liter na duże,
`int tolower(int c)` – zamiana dużych liter na małe,
`int is...(int c)` – funkcje rozpoznania z tabeli 11.1.

Użycie tych funkcji wymaga włączenia pliku **ctype.h** poleceniem #*include* <*ctype.h*>. Argumentem każdej z tych funkcji jest kod znaku. Jeśli argumentem funkcji *toupper* jest mała litera to wynikiem jest duża litera. Jeśli argumentem jest inny znak, to wynikiem jest ten sam znak bez konwersji. Podobnie funkcja *tolower* zwraca znak małej litery lub znak bez konwersji.

Funkcje sprawdzenia dają w wyniku zero, gdy dany znak nie jest z określonej grupy lub liczbę różną od zera, gdy należy do tej grupy. Wykaz funkcji sprawdzania zawiera tabela 11.1.

Przykłady

Aby w odpowiedziach typu *TAK*/*NIE* uprościć analizę naciskanych klawiszy *T* lub *N* i porównywać je tylko z dużymi literami, można użyć następujących instrukcji

```
do {z=toupper(getch());} while(z!='T' && z!='N');
```

Każda z poniższych dwu instrukcji zamieni małe litery na duże w tekście wskazywanym przez *q*

```
while(*q) *q++ = toupper(*q);
for(i=0; q[i]; i++) q[i]=toupper(q[i]);
```

Poniższe instrukcje zastąpią w tekście wskazywanym przez *P* znaki sterujące spacjami, a następnie wypiszą z tego tekstu cyfry.

```
for(q=P; *q; q++) if(iscntrl(*q)) *q = ' ';
for(q=P; *q; q++) if(isdigit(*q)) putch(*q++);
```

Tabela 11.1. Funkcje sprawdzania znaków

Funkcja	Nazwa grupy	Znaki należące do grupy
isalnum	alfanumeryczne	'0'..'9', 'A'..'Z', 'a'..'z'
isalpha	litery	'A'..'Z', 'a'..'z'
islower	małe litery	'a'..'z'
isupper	duże litery	'A'..'Z'
isdigit	cyfry	'0'..'9'
isxdigit	heksagonalne	'0'..'9', 'A'..'F', 'a'..'f'
isspace	białe znaki	spacja, '\t', '\r', '\n', '\v', '\f'
iscntrl	sterujące	o kodach 0..31, 127
isprint	drukowalne	o kodach 32..126
isgraph	widoczne	drukowalne bez białych
ispunct	interpunkcyjne	'!'..'/', ':'..'@', '['..'`', '{'..'~'
isascii	znaki ASCII	o kodach 0..127

Pytania i zadania

11.1. Napisz funkcję, która w danym tekście:
a) zamieni wszystkie litery na duże, b) zamieni pierwsze litery na duże.

11.2. Napisz funkcję, która sprawdzi, czy dany tekst zawiera:
a) poprawną nazwę w sensie języka C, b) poprawną liczbę rzeczywistą.

11.2. Działanie na tekstach

Funkcje operujące na tekstach to funkcje kopiowania, dopisywania i porównywania tekstów oraz funkcje zliczania i wyszukiwania znaków w tekście. Użycie tych funkcji wymaga włączenia pliku **string.h**.

Funkcje kopiowania i dopisywania to

```
char *strcpy(char *dest, char *source)          – kopiowanie,
char *strncpy(char *dest, char *source, int n)
char *strcat(char *dest, char *source)          – dopisanie.
char *strncat(char *dest, char *source, int n)
```

Funkcje te przepisują tekst wskazywany przez *source* do obszaru wskazywanego przez *dest* od początku tego obszaru w przypadku kopiowania lub za tekst istniejący w tym obszarze w przypadku dopisywania. Przepisywany jest cały tekst (*strcpy,*

strcat) lub maksymalnie *n* znaków (*strncpy, strncat*). Funkcje te dają w wyniku wskazanie *dest*. Funkcja *strncpy* kopiuje zawsze *n* znaków. Jeśli tekst kopiowany ma *n* lub więcej znaków, to kopiowane jest *n* znaków bez znaku zerowego. Jeśli tekst jest krótszy, to dopisywane są znaki zerowe. Funkcje *strcat* i *strncat* zawsze łączą oba teksty w *dest* w jeden tekst ograniczony znakiem '\0'.

Przykładowe instrukcje połączą w *buf* trzy teksty zawarte w tablicach *buf*1 i *buf*2

```
buf3=(char*)malloc(strlen(buf1) + strlen(buf2) + 1);
if(buf3 != NULL)      {strcpy(buf3, buf1);
                       strcat(buf3, buf2);}
```

Parametr *dest* musi wskazywać na tablicę znakową (bufor) takich rozmiarów, aby mogła ona pomieścić cały wynikowy tekst wraz z ogranicznikiem '\0'. Fatalnym błędem jest wskazanie *dest* na zbyt małą tablicę lub gdy jest ono wskazaniem pustym *NULL*.

Funkcje zliczania znaków to funkcje typu *size_t* (*unsigned*)

```
size_t strlen(char *str)
size_t strcspn(char *str, char *str2)
```

Wynikiem jest liczba znaków tekstu wskazywanego przez *str*: do końca (bez '\0') albo do pierwszego znaku występującego w tekście wskazywanym przez *str*2.

Funkcje porównania tekstów

```
int strcmp(char *str1, char *str2);
int stricmp(char *str1, char *str2);
int strncmp(char *str1, char *str2, int n);
```

Porównywane mogą być całe teksty (*strcmp, stricmp*) lub maksymalnie *n* pierwszych znaków (*strncmp*). Funkcja *strcmp* porównuje kody znaków traktowane jako liczby typu *signed char*. Tak więc litera *Z* jest przed literą *a*, a ta przed literą *z*. Podobnie działa funkcja *stricmp*, z tym że nie rozróżnia małych i dużych liter. Wynikiem porównania jest wartość:

strcmp(str1, str2)
stricmp(str1, str2)

$$\begin{cases} < 0 & \text{gdy } str1 \text{ jest przed } str2, \\ = 0 & \text{gdy } str1 \text{ i } str2 \text{ są jednakowe}, \\ > 0 & \text{gdy } str1 \text{ jest po } str2. \end{cases}$$

Funkcje wyszukiwania znaków lub podtekstów w zadanym tekście

```
char *strchr(char *str, int c)          – pierwszy znak c,
char *strrchr(char *str, int c)         – ostatni znak c,
char *strpbrk(char *str, char *str2)    – pierwszy znak ze str2,
char *strstr (char *str, char *str2)    – podtekst str2.
```

Wszystkie te funkcje poszukują zadanego obiektu w tekście wskazywanym przez *str*. Wynikiem jest wskazanie znalezionego obiektu, to znaczy wskazanie: pierwszego (*strchr*) albo ostatniego (*strrchr*) wystąpienia znaku *c*, pierwszego wystąpienia dowolnego ze znaków występujących w tekście *str*2 (*strpbrk*) lub pierwszego wystąpienia podtekstu *str*2 (*strstr*). Jeśli zadany obiekt nie zostanie znaleziony, funkcje zwracają w wyniku wskazanie puste NULL.

Na przykład poniższe instrukcje nadają zmiennej *q*1 wskazanie do litery *r* w tekście „Borland C++", a zmiennej *q*2 wskazanie puste NULL, ponieważ w tekście „Borland C++" nie występuje żadna z liter tekstu „mysz".

```
q1=strpbrk("Borland C++", "karty");
q=strpbrk("Borland C++", "mysz");
```

Funkcje konwersji tekstów

```
char *stlwr(char *str)    – zamiana liter na małe,
char *strupr(char *str)   – zamiana liter na duże.
```

w podanych tekstach zamienią wszystkie duże litery na małe (*stlwr*) lub małe na duże (*strupr*). Wynikiem obu funkcji jest wskazanie konwertowanego tekstu *str*.

Pytania i zadania

11.3. Napisz odpowiednik funkcji *strncpy*, który zawsze stworzy tekst zakończony znakiem ogranicznika '\0'.

11.4. Napisz funkcję, która w danym tekście zastąpi pojedynczymi spacjami:
a) znaki tabulacji '\t', b) znaki '\t', '\n' i '\f'.

11.3. Konwersje liczbowo tekstowe

Funkcje konwersji tekstów na liczby i liczb na teksty są opisane w pliku **stdlib.h**. Funkcja *sprintf* jest opisana w pliku **stdio.h**. Funkcja *atof* jest opisana zarówno w pliku **stdlib.h**, jak i w **math.h**.

Funkcje konwersji tekstu na liczbę

```
int atoi(char *tekst)      – konwersja do typu int
long atol(char *tekst)     – konwersja do typu long,
double atof(char *tekst)   – konwersja do typu double.
```

Funkcje konwersji liczby na tekst

```
char *itoa(int N, char *buf, int podst)
char *ltoa(long N, char *buf, int podst)
char *ultoa(unsigned long N,char *buf,int podst)
```

Funkcje te konwertują wartości kolejno typu *int*, *long* oraz *unsigned long*. Każda funkcja konwersji liczby na tekst tworzy tekstowy zapis liczb w systemie o zadanej podstawie *podst* i umieszcza ten tekst w buforze wskazywanym przez *buf*. Bufor ten musi mieć taki rozmiar, aby pomieścić cały utworzony tekst. Wynikiem jest wskazanie *buf*.

Na przykład następująca instrukcja wyprowadzi liczbę całkowitą (typu *int*) *k* w postaci binarnej. Bufor *txt* powinien być zdefiniowany jako tablica tekstowa zawierająca co najmniej 17 elementów.

```
printf(itoa(k, txt, 2));
```

Redagowanie tekstu z liczbami – funkcja sprintf

Szerokie możliwości tworzenia tekstu z liczbami daje funkcja *sprintf*

```
int sprintf(char *bufor, char *format, ...);
```

Funkcja *sprintf* wyprowadza tekst do wskazanego pierwszym argumentem *bufor* bufora tekstowego podobnie jak *fprintf* do strumienia. Tekst zakończony jest znakiem zerowym, a całość nie może przepełnić podanego bufora. Funkcji *sprintf* często używa się do przygotowania tekstów wyprowadzanych w trybie graficznym, jak pokazano w poniższym przykładzie

```
char B[30];
double x;
int i;
for(i=1; i<6; i++)
  { x=1.5*i;
    sprintf(B,"I=%d  X=%5.1lf", i, x);
    outtextxy(16, 12*i, B);
  }
```

Pytania i zadania

11.5. Napisz fragment programu, który zinterpretuje argumenty wywołania programu (*char *argv*[]) jako liczby: a) typu *int*, b) typu *double*.

Zadania laboratoryjne

11.6. Napisz program, który analizując argument *argv*[0] wypisze, z jakiego dysku i z jakiej kartoteki został wywołany i jaką ma nazwę.

11.7. Napisz program, który podaną liczbę dziesiętną wyprowadzi w postaci liczby o zadanej podstawie.

11.8. Napisz program, który posortuje wyrazy w kolejności alfabetycznej utożsamiając małe i duże litery.

11.9. Napisz program, który wczytując dowolny zbiór tekstowy, sporządzi dla tego pliku alfabetyczny słownik wyrazów.

12. Wybrane techniki programowe

12.1. Obsługa błędów

W języku C błędy na ogół nie wstrzymują wykonania programu. Za wykrycie błędów i reakcje programu na błędy odpowiada programista. Na przykład programista musi przewidzieć, że wykonanie funkcji *fopen* może zakończyć się niepowodzeniem. Musi on zatem umieścić w programie instrukcje, które sprawdzają poprawność wykonania takich funkcji i zaprogramować reakcję programu na wystąpienie błędów. Programista ma do dyspozycji odpowiednie narzędzia.

W pliku **stdlib.h** zdefiniowano i zainicjowano następujące zmienne globalne:

```
int sys_nerr;                  – liczba pozycji tablicy sys_errlist[ ],
char *sys_errlist[sys_nerr];   – tablica tekstów opisu błędów,
int errno;                     – numer ostatniego błędu.
```

Operacje wejścia i wyjścia w przypadku błędu zapisują jego numer do zmiennej *errno*. W pozostałych przypadkach musi to zrobić programista. Odpowiednie numery błędów zdefiniowano w pliku **errno.h**. Element *sys_errlist*[*nr*] wskazuje na tekst opisujący błąd o numerze *nr*. Na przykład poniższy program wyprowadzi wszystkie komunikaty błędów z tablicy *sys_errlist*.

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

void main()
  { int i;
    for(i=0; i<sys_nerr; i++)
          printf("\n%3d  %s", i, sys_errlist[i]);
  }
```

W pliku **stdio.h** są prototypy funkcji:

```
int ferror(FILE *stream)          – wartość wskaźnika błędu pliku,
void perror(const char *string)   – wyprowadzenie komunikatu błędu,
char *strerror(int nr)            – wskazanie komunikatu błędu.
```

Funkcja *ferror* daje w wyniku 1, gdy jest ustawiony wskaźnik błędu podanego pliku lub 0, gdy wskaźnik nie jest ustawiony.

Funkcja *perror* wyprowadza do pliku **stderr** komunikat o błędzie, którego numer został zapamiętany w zmiennej *errno*. Rezultat jest podobny do użycia instrukcji

```
fprintf(stderr, "%s:%s\n", string, sys_errlist[errno]);
```

W przypadku zaniechania dalszych obliczeń po stwierdzeniu błędu należy pozamykać otwarte pliki i zwolnić zbędnie zaalokowaną pamięć. Należy pamiętać, że fatalnym błędem jest zwalnianie pamięci, która nie została przydzielona.

Przykład obsługi błędów w funkcji, która alokuje tablice i otwiera pliki:

```
{ int i, j, n, m;
  double **A
  FILE *fp1, *fp2;
  errno=ENOMEM;
                  // ENOMEM – zdefiniowana w errno.h (brak pamięci)
  if((fp1=fopen("nazwa1", "r"))==NULL)  goto et1;
  if((fp2=fopen("nazwa2", "w"))==NULL)  goto et2;
  if(fscanf(fp1, "%d%d", &n, &m) < 2)   goto et3;
  A=(double**)calloc(n, sizeof(double*));
  if(!A) goto et4;
  for(i=0; i<n; i++)
    if((A[i]=(double*)calloc(m, sizeof(double)))==NULL)
                                                goto et5;
  . . . . . .
  if(fprintf(fp2, . . .)==EOF) goto et6;
  . . . . . .
  errno=0;
et6:i=n;
et5:for(i--; i>=0; i--) free(A[i]);
  free(A);
et4:et3:fclose(fp2);
et2:fclose(fp1);
et1:if(errno) perror("PRZYKLAD");
  return( . . . );
}
```

Pytania i zadania

12.1. Jak zmieni się kolejność ostatnich (zaetykietowanych) instrukcji w przykładowej funkcji, gdy zbiór *nazwa*2 będzie otwierany po zaalokowaniu tablicy *A*,

12.2. Napisz z uwzględnieniem obsługi błędów funkcję *Czytaj*, która otworzy plik binarny z liczbami typu *float*, określi ile jest tych liczb, wczyta te liczby do zaalokowanej tablicy i zamknie ten plik. Wynikiem funkcji będzie wskazanie zaalokowanej tablicy. Ilość liczb należy przekazać za pomocą parametru. Gdzie należy zwolnić tak zaalokowaną tablicę?

12.2. Obsługa plików dyskowych

Następujące funkcje opisane w **dir.h** umożliwiają posługiwanie się katalogami:

```
int setdisk(int drive)          – zmiana aktualnego dysku,
int chdir(const char *path)     – zmiana aktualnej kartoteki,
int mkdir(const char *path)     – utworzenie katalogu,
int rmdir(const char *path)     – skasowanie katalogu.
```

W **stdio.h** oraz w **dos.h** są prototypy funkcji usuwania i przemianowywania plików :

```
int unlink(const char *name)    – usunięcie pliku,
int remove(const char *name)    – usunięcie pliku,
int rename(const char *stara,   – zmiana nazwy pliku.
           const char *nowa)
```

Nie można usuwać plików przeznaczonych tylko do odczytu oraz nie można usuwać ani przemianowywać plików otwartych.

Za pomocą funkcji *rename* można nie tylko przemianować plik, ale też można przenieść go do innego katalogu na tym samym dysku.

Na przykład, poniższe instrukcje utworzą w aktualnym katalogu podkatalog o nazwie *DANE* i przeniosą do niego plik *dane*1.*dat* z tą samą nazwą oraz przeniosą plik *ddd* , nadając mu nazwę *dane*2.*dat*. Następnie ten nowy katalog stanie się katalogiem aktualnym.

```
mkdir("DANE");
rename("dane1.dat", "DANE\\dane1.dat");
rename("ddd", "DANE\\dane2.dat");
chdir("DANE");
```

Poniższy program usunie z pliku *dane.dat* zbędne spacje. Aby zapewnić ochronę pliku w czasie edycji, zostanie najpierw utworzony tymczasowy plik *dane.tmp*. Następnie plik *dane.dat* zostanie usunięty, a plik tymczasowy zostanie przemianowany na *dane.dat*. Dla wygody i dla uniknięcia pomyłek w nazwach plików użyto tych nazw jednokrotnie w definicji wskaźników *dane* i *tmp*. Program wzbogacono o obs³ugê b³êdów.

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
  { char z1, z2, *dane="dane.dat", *tmp="dane.tmp";
    FILE *fp1, *fp2;
    if((fp1=fopen(dane, "r"))==NULL)
                        {perror(dane); goto et1;}
    if((fp2=fopen(tmp, "w"))==NULL)
                        {perror(tmp); goto et2;}
    errno=0;
    for(z2=0; !feof(fp1); z2=z1)
        if((z1=fgetc(fp1))!=' ' || z2!=' ') fputc(z1, fp2);
    fclose(fp2);
    if(errno) remove(tmp);
et2:fclose(fp1);
    if(!errno)  {remove(dane);  rename(tmp, dane); }
et1:return;
}
```

Pytania i zadania

12.3. Jak dokonać edycji pliku zachowując jego poprzednią zawartość w pliku o tej samej nazwie, ale z rozszerzeniem .BAK ? Napisz odpowiednie instrukcje.

12.4. W jaki sposób program, niezależnie od tego jak nazywa się jego plik wykonywalny, może usunąć się z dysku?

12.3. Obsługa zegara i pomiar czasu

W pliku **time.h** zdefiniowano typ **time_t** = **unsigned long** oraz kilka funkcji przekształcania informacji o bieżącym czasie. Przykładowe funkcje to:

```
long clock(void)            – czas wykonywania się programu,
char          *ctime(time_t – przekształcenie daty i czasu do postaci zna-
*clock)                       kowej,
```

`time_t *clock)` `time(time_t` – liczba sekund, które upłynęły od 1.01.1970 r.

W pliku **dos.h** opisano funkcje, które umożliwiają odczytanie i ustawienie systemowej daty i systemowego czasu oraz funkcje, które wstrzymują wykonanie programu:

`void setdate(struct date *date)`	– ustawienie daty systemowej,
`void settime(struct time *time)`	– ustawienie czasu systemowego,
`void getdate(struct date *date)`	– odczytanie daty systemowej,
`void gettime(struct time *time)`	– odczytanie czasu systemowego,
`void delay(unsigned time)`	– wstrzymanie programu na *time* milisekund,
`void sleep(unsigned sec)`	– wstrzymanie programu na *sec* sekund.

Struktury *date* i *time* są zdefiniowane w pliku **dos.h**

```
struct date {
  int    da_year;          // rok
  char   da_day;           // dzień
  char   da_mon;};         // miesiąc

struct time {
  unsigned char ti_min,    // minuty
                ti_hour,   // godziny
                ti_hound,  // setne części sekundy
                ti_sec;};  // sekundy
```

Przykłady

Funkcja *clock* podaje czas w umownych jednostkach. Aby wyrazić ten czas w sekundach, należy podzielić go przez stałą CLK_TCK, zdefiniowaną w pliku *time.h* jako CLK_TCK = 18.2.

Czas, jaki upłynął od początku wykonywania się programu, wyprowadzi instrukcja

```
printf("Czas obliczen: %.3lf sekund\n", clock()/CLK_TCK);
```

Po zdefiniowaniu zmiennej *time_t czas* aktualną datę w języku angielskim wyprowadzić można instrukcjami:

```
time(&czas);
printf(ctime(&czas));
```

lub

```
printf(ctime(time(&czas)));
```

Datę w postaci *dd.mm.rrrr* wyprowadzą instrukcje:

```
struct date D;
getdate(&D);
printf("%2d.%02d.%4d\n", D.da_day, D.da_mon, D.da_year);
```

Kolejne funkcje wstrzymują wykonanie programu na 0,25 sekundy i na 7 sekund

```
delay(250);
sleep(7);
```

Pytania i zadania

12.5. Jak określić czas wykonywania krótkich instrukcji, np. z dokładnością do 10 mikrosekund?

12.6. Podaj praktyczny przykład zastosowania funkcji *getdate* i *gettime*.

12.4. Dźwięk

Najprostszym sposobem wygenerowania krótkotrwałego sygnału dźwiękowego jest wyprowadzenie na monitor znaku „*bel*" o kodzie '\7' lub symbolu '*a*'. Na przykład instrukcja

```
printf("Podaj dane: \a");
```

wyprowadzi tekst „Podaj dane: " i wygeneruje krótki pisk.

Do generowania dźwięków o zadanej wysokości tonów służą funkcje opisane w pliku **dos.h**:

`void sound(unsigned freq)` – ton o częstotliwości *freq* Hz,
`void nosound(void)` – wyłączenie tonu.

Na przykład, ton o częstotliwości 1600 Hz i czasie trwania 1,2 s wygenerują instrukcje:

```
sound(1600);
delay(1200);
nosound();
```

Pytania i zadania

12.7. Napisz instrukcje, które będą cyklicznie aż do naciśnięcia klawisza *Esc*, generować sygnał SOS (trzy dźwięki krótkie, trzy długie i trzy krótkie).

12.5. Dostęp do pamięci i do portów

W pliku **dos.h** zamieszczono opisy funkcji, które umożliwiają odczyt bajta lub słowa z portu lub pamięci oraz zapis bajta lub słowa do portu lub pamięci. Są to funkcje

```
unsigned inport(unsigned nr)                    – odczyt słowa z portu,
unsigned char inportb(unsigned nr)              – odczyt bajta z portu,
void outport(unsigned nr,unsigned word)         – zapis słowa do portu,
void outportb(unsigned nr,unsigned char byte)
                                                – zapis bajta do portu,
int peek(unsigned seg,unsigned off)                   – odczyt słowa,
char peekb(unsigned seg,unsigned off)                 – odczyt bajta,
void poke(unsigned seg,unsigned off,int value)        – zapis słowa,
void pokeb(unsigned seg,unsigned off,char value)      – zapis bajta.
```

Funkcje *inport* i *inportb* dają w wyniku słowo i bajt, odczytane z portu o numerze *nr*. Funkcje *outport* i *outportb* zapisują słowo i bajt do portu o numerze *nr*. Na przykład, ustawienie dwóch najmłodszych bitów w porcie o adresie 0x61 włącza głośnik, a wyzerowanie tych bitów wyłącza głośnik. Dokonują tego instrukcje:

```
outport(0x61, inport(0x61) | 3);
outport(0x61, inport(0x61) & 0xFC);
```

Funkcje *peek* i *peekb* dają w wyniku słowo i bajt odczytane z pamięci spod adresu określonego przez *seg* i *off*. Funkcje *poke* i *pokeb* zapisują słowo i bajt do pamięci. Wartości *seg* i *off* są tu kolejno segmentem i offsetem wskaźnika słowa lub bajta. Tak więc odczyt i zapis dotyczą adresu

```
16 * seg + off.
```

Status klawiatury odczyta każda z poniższych instrukcji

```
status=peek(0, 0x0417);
```

```
status=peek(0x0041, 0x0007);
```

Kolejne bity statusu, od najm³odszego, sygnalizuj¹ naciœniêcie klawiszy: 0 – *prawy Shift*, 1 – *lewy Shift*, 2 – *Ctrl*, 3 – *Alt*, 8 – *lewy Ctrl*, 9 – *lewy Alt*, 12 – *Scroll Lock*, 13 – *Num Lock*, 14 – *Caps Lock* i 15 – *Insert* oraz w³¹czenie: 4 – *Scroll Lock*, 5 – *Num Lock*, 6 – *Caps Lock* i 7 – *Insert*. Bity ustawione oznaczają klawisz naciśnięty lub tryb włączony. Tak więc np. instrukcje sprawdzają kolejno:

```
peek(0, 0x0417) & 1;      – czy jest naciśnięty prawy Shift,
peek(0, 0x0417) & 8;      – czy jest naciśnięty dowolny Alt,
peek(0, 0x0417) & 0x20;   – czy jest włączony tryb Num Lock.
```

Natomiast:

```
poke(0, 0x0417, peek(0, 0x0417) | 0x20);   – włącza Num Lock,
poke(0, 0x0417, peek(0, 0x0417) & 0x6F);   – wyłącza Num Lock.
```

Pytania i zadania

12.8. Napisz instrukcje, które: a) włączą tryb *Caps Lock*, b) wyłączą tryb *Caps Lock*, c) włączą tryb *Insert*.

12.9. Napisz instrukcje, które określą stan klawiszy *Ctrl*, i *Alt* z rozróżnieniem prawego i lewego klawisza.

12.10. Napisz funkcję, która zmieni atrybuty ekranu tekstowego w zadanym prostokącie (np. dokona inwersji kolorów, zmieni kolor tła itp.), dokonując zapisu bezpośrednio do pamięci ekranu.

12.6. Wykorzystanie przerwań

W pliku **dos.h** opisano funkcje generujące przerwania programowe:

```
int int86(int Nr, union REGS *in, union REGS *out);
int int86x(int Nr, union REGS *in, union REGS *out,
                                   struct SREGS *sr);
int intdos(union REGS *in, union REGS *out);
int intdosx(union REGS *in, union REGS *out,
                                   struct SREGS *sr);
void intr(int Nr, struct REGPACK *reg);
```

oraz funkcje pobrania i zmiany wektora przerwań:

```
void interrupt (*getvect(int Nr))();
void setvect(int Nr, void interrupt (*isr)());
```

Funkcje *int*86, *int*86*x* generują przerwanie programowe o numerze *Nr*. Funkcje *intdos* i *intdosx* generują przerwanie programowe o numerze 0x21. Unia *REGS* oraz struktury *SREGS* i *REGPACK* są zdefiniowane w **dos.h**

```
union REGS {
 struct {unsigned ax, bx, cx, dx, si, di, cflag, flags;} x;
 struct {unsigned char al, ah, bl, bh, cl, ch, dl ,dh; } h;
};

struct SREGS {unsigned es, cs, ss, ds;};

struct REGPACK{
 unsigned r_ax, r_bx, r_cx, r_dx, r_bp, r_si, r_di, r_ds,
             r_es, r_flags;
};
```

Nazwy składowych powyższych struktur ściśle korespondują z nazwami rejestrów mikroprocesora. I tak np. *ax* lub *r_ax* oznacza 16-bitowy rejestr *AX* procesora, a *al* i *ah* oznaczają kolejno młodszy i starszy bajt tego rejestru. Funkcje załadowują rejestry procesora danymi zawartymi w unii *in* (oraz w strukturze *sr* zapamiętując rejestr *DS* procesora – funkcje *int*86*x* i *intdosx*), generują przerwanie, a następnie kopiują dane zawarte w rejestrach procesora do unii *out* (i odtwarzają rejestr *DS*). Jako wynik funkcje te zwracają wartość rejestru *AX* procesora.

Funkcja *intr* kopiuje dane ze struktury *reg* do rejestrów procesora, generuje przerwanie i kopiuje dane z rejestrów procesora do struktury *reg*.

Funkcja *getvect* daje w wyniku dalekie wskazanie do funkcji obsługi przerwania o numerze *Nr*. Funkcja *setvect* dokonuje takiej zmiany wektora przerwań, że w odpowiedzi na przerwanie o numerze *Nr* jest wykonywana funkcja wskazywana przez *isr*.

Funkcje przewidziane do obsługi przerwań powinny być definiowane z modyfikatorem **interrupt**. Takie funkcje zapamiętują automatycznie na stosie nie tylko rejestry *SI*, *DI* i *BP* procesora, ale też rejestry *AX*, *BX*, *CX*, *DX*, *ES* i *DS*, aby w epilogu odtworzyć stan procesora sprzed wywołania.

Zmiana wektora przerwań (*setvect*) polegająca na zastąpieniu systemowej funkcji obsługi danego przerwania przez inną funkcję zdefiniowaną z modyfikatorem *interrupt* powinna być poprzedzona zapamiętaniem (*getvect*) systemowego wektora, aby można było odtworzyć oryginalną obsługę przerwań.

Przykładowy program wypisuje na ekranie kolejne liczby i równocześnie gra melodię. Melodia jest grana z szybkością niezależną od szybkości obliczeń. Systemowa obsługa przerwań zegarowych o numerze 0x1C zostaje zamieniona na obsługę funkcją *zegar*. W ciągu 1 sekundy zegar systemowy generuje ok. 18 przerwań o numerze 0x1C. Funkcja *zegar* zmienia wysokość tonu co dziewiąte wywołanie, czyli co ok. pół sekundy.

```
  #include <conio.h>
  #include <dos.h>
  int
T[]={440,587,698,659,587,698,587,659,587,466,523,440,440,0};
  int F=1;

  void interrupt far zegar(void)              // funkcja obsługi przerwań
    { static int i=0, n=0;
      if(i) {i--; return;}
      i=8;
      F=T[n++];
      if(F) sound(F); else nosound();
    }

  void main(void)
    { int n=1, nri=0x1C;
      void interrupt(far *st)(void);          // definicja wskaźnika st
      st=getvect(nri);                        // zapamiętanie wektora systemowego
      setvect(nri, zegar);                    // zmiana wektora
      do  {  cprintf("%8o\n\r", n++);
          } while(F);
      setvect(nri, st);                       // odtworzenie wektora systemowego
    }
```

12.7. Predefiniowane zmienne

W plikach *stdlib.h* oraz *dos.h* znajdują się między innymi definicje niżej opisanych zmiennych globalnych:

errno – (*int*) numer błędu zgłaszany przez funkcję, w której opisie wymieniono tę zmienną,

_8087 – (*int*) numer koprocesora: 0 – brak koprocesora
1 – 8087
2 – 80387
3 – 80387,

_fmode – (*int*) domyślny tryb przetwarzania pliku (domyślnie *O_TEXT*),

_heaplen – (*unsigned*) rozmiar bliskiej strefy w bajtach (tylko w modelach: *TINY*, *SMALL* oraz *MEDIUM*), domyślna wartość równa zero zapewnia maksymalny rozmiar strefy,

_stklen – (*unsigned*) rozmiar stosu (domyślnie 4 kB); aby zmienić rozmiar stosu, należy dokonać podstawienia na zewnątrz wszystkich funkcji (na poziomie globalnym),

_version – (*unsigned*) numer wersji systemu DOS: numer nadrzędny w młodszym bajcie, natomiast numer podrzędny w starszym bajcie,

_psp – (*unsigned*) adres (numer segmentu) startowy programu w pamięci.

Przykład 1

Aby uzyskać maksymalny rozmiar stosu (np. w modelu *LARGE*), należy na poziomie globalnym zdefiniować

```
extern unsigned _stklen=0xFFE0;
```

Można wówczas otrzymać ostrzeżenie od linkera o konflikcie z istniejącym podstawieniem wartości domyślnej, ale przyjęta zostanie wartość zdefiniowana w programie źródłowym

Przykład 2

Aby uczynić program rezydentnym, należy posłużyć się funkcją *keep* o nagłówku

```
void keep(unsigned char status, unsigned size);
```

gdzie parametr *size* określa rozmiar programu w paragrafach. Zmienna *_psp* jest numerem paragrafu (segmentu), w którym zaczyna się program. Szczyt stosu jest zarazem końcem programu. Rejestr *SS* i zmienna *_SS* zawierają numer paragrafu szczytu stosu. Rejestr *SP* i zmienna *_SP* zawierają rozmiar stosu w bajtach. Tak więc rozmiar programu w paragrafach wyznacza się ze wzoru

```
rozmiar_programu_w_paragrafach = _SS + (_SP/16) - _psp;
```

Wywołanie funkcji

```
keep(0, rozmiar_programu_w_paragrafach);
```

uczyni program rezydentnym jeżeli był on skompilowany z wyłączoną kontrolą przepełnienia stosu *Test Stack Overflow*.

12.8. Preprocesor

Preprocesor modyfikuje tekst źródłowy do kompilacji. Polecenia preprocesora zaczynają się znakiem #. Najczęściej używanymi poleceniami są *#include* oraz *#define*. Na przykład:

```
#include <stdio.h>
#define  Nmax 250
```

Polecenie *#include* wstawia w swoje miejsce zawartość podanego pliku. Polecenie *#define* przedefiniowuje ciągi znaków. Może ono też definiować identyfikatory oraz makrodefinicje. Na przykład:

```
#define __STDIO_H
#define max(x,y)   ((x)>(y)?(x):(y))
```

Definicje można usuwać poleceniem *#undefine*. Na przykład, identyfikator *__STDIO_H* jest zdefiniowany między wyżej podanym poleceniem *#define* a poleceniem

```
#undefine __STDIO_H
```

Do sprawdzania, czy podany identyfikator jest zdefiniowany, służą polecenia *#ifdef*, *#ifndef* oraz operator *defined*, który daje w wyniku wartość 1, gdy identyfikator jest zdefiniowany lub 0, gdy nie jest. Identyfikator może (ale nie musi) być ujęty w nawiasy okrągłe np.:

```
#if !defined(__STDIO_H)
#include <stdio.h>
#endif
```

Powyższe polecenia włączą do programu plik *stdio.h*, jeśli identyfikator *__STDIO_H* nie jest zdefiniowany. Powyższy przykład ilustruje użycie polecenia *#if*. Jeżeli wyrażenie w tym poleceniu jest prawdziwe, to analizowane są wszystkie linie programu od *#if* do *#endif*. W przeciwnym razie linie te są pomijane. W poleceniach *#ifdef* oraz *#ifndef* zamiast warunku podaje się identyfikator. Następujące dalej linie są analizowane, gdy ten identyfikator jest (*#ifdef*) lub nie jest (*#ifndef*) zdefiniowany. Warunkowe włączenie pliku *stdio.h* można zrealizować poleceniami

```
#ifndef __STDIO_H
#include <stdio.h>
#endif
```

Wszystkie trzy postacie polecenia *#if* można rozbudować za pomocą poleceń *#else* oraz *#elif*. W tym przypadku, zależnie od warunku, analizowane są linie między *#if* a *#else* albo między *#else* a *#endif*. Polecenie *#elif* jest złożeniem *else if* i poprzedza warunek, który steruje analizą dalszych linii.

Polecenia *#if* mogą się wzajemnie zagłębiać. Na przykład, w zależności od modelu pamięci należy zdefiniować *NULL* jako 0 (dwubajtowe) lub 0L (czterobajtowe). Definicja ta powinna wystąpić tylko raz, niezależnie od tego, ile razy będzie włączany plik *_null.h*. A oto jak w tym pliku rozwiązano powyższy problem:

```
#ifndef NULL
# if defined(__TINY__)||defined(__SMALL__)||defined(__MEDIUM__)
#    define NULL    0
#  else
#    define NULL    0L
#  endif
#endif
```

Dodajmy, że w zależności od modelu pamięci automatycznie jest definiowany jeden z identyfikatorów: *__TINY__*, *__SMALL__*, *__MEDIUM__*, *__COMPACT__*, *__LARGE__*, *__HUGE__*.

Jeśli program jest kompilowany kompilatorem obiektowym, to zdefiniowany jest identyfikator *__cplusplus*.

Jeśli jest to program wymagający środowiska Windows, to zdefiniowany jest identyfikator *_Windows*.

Należy zwrócić uwagę, aby nie włączać wielokrotnie tych samych plików i nie powtarzać zawartych w nich definicji. Powyżej pokazano warunkowe włączanie pliku *stdio.h*. Aby ta konstrukcja działała, w pliku *stdio.h* należy zdefiniować identyfikator *__STDIO_H*. Zwykle każdy plik nagłówkowy definiuje identyfikator podobny do nazwy tego pliku. W takim pliku na początku sprawdza się, czy ten identyfikator jest już zdefiniowany. Jeśli tak, to pozostałe linie pliku pomija się. Inne pliki pomijają polecenie *#include*, jeśli odpowiedni identyfikator jest już zdefiniowany. Stanowi to system podwójnego zabezpieczenia. Poniższe przykłady fragmentów plików *stdio.h* oraz *_defs.h* ilustrują, jak zabezpieczać się przed powielaniem włączania plików.

Plik _defs.h

```
#if !defined(___DEFS_H)
```

```
   #define ___DEFS_H          // warunkowa definicja
   . . .                      // warunkowo włączona dalsza część pliku
#endif                        // koniec pierwszego if
```

Plik stdio.h

```
#ifndef __STDIO_H
   #define __STDIO_H          // warunkowa definicja

   #if !defined(___DEFS_H)
      #include <_defs.h>      // warunkowe włączenie pliku _defs.h
   #endif

   #ifndef NULL
      #include <_null.h>      // warunkowe włączenie pliku _null.h
   #endif

   . . .                      // warunkowo włączona dalsza część pliku

#endif                        // koniec pierwszego ifndef
```

Zadania laboratoryjne

12.11. Napisz program, który utworzy adresową bazę danych i będzie ją obsługiwać (umożliwi dopisywanie, korektę, kasowanie, sortowanie i drukowanie adresów). Program napisz tak, aby ani błędy operatora, ani awaria zasilania nie uszkadzały istniejącej bazy.

12.12. Napisz program, który w odpowiedzi na naciskanie klawiszy będzie generować:
a) tony o wysokości zależnej od klawisza,
b) naciśnięty znak alfabetem Morse'a.

12.13. Napisz program, który będzie identyfikować naciskane klawisze *Shift*, *Ctrl*, *Alt*, prawe i lewe.

12.14. Napisz program, który spowoduje migotanie lampki *Num Lock* z częstotliwością dwa razy na sekundę.

12.15. Napisz program, który wykorzystując przerwanie 0x1C, spowoduje wyświetlanie czasu w prawym górnym rogu ekranu. Podczas naciskania prawego klawisza *Shift* w miejsce czasu powinna być wyświetlana data.

13. Ćwiczenia laboratoryjne

Ćwiczenie 1. Edytor i kompilator

Cel: Przygotowanie własnego środowiska do edycji i kompilacji, edycja i kompilacja przykładowych programów oraz nauka korzystania z pomocy „help".

W ćwiczeniu należy:

a) Przygotować dyskietki (lub katalogi) robocze i przekopiować do nich pliki **tcdef.dpr**, **tcdef.dsk** oraz **tcconfig.tc** z katalogu ...\BC\BIN.
b) Uruchomić edytor z roboczego katalogu (lub dyskietki), ustawić wymagane opcje i zapisać własną konfigurację.
c) Napisać i uruchomić przykładowe programy. Możliwie jak najwcześniej nadać plikowi docelową nazwę. Pisanie programu zacząć od napisania początkowych i końcowych linii programu, tak aby można było na bieżąco kompilować i uruchamiać program napisany częściowo.

Proponowane zadania: 1.8, 1.9

Ćwiczenie 2. Debugger

Cel: Poznanie metod testowania programów, śledzenie wykonania krok po kroku, zakładanie i wykorzystanie pułapek, obserwowanie wartości zmiennych oraz modyfikowanie tych wartości.

Proponowane zadania: 1.10–1.14

Ćwiczenie 3. Obsługa ekranu i klawiatury

Cel: Przyswojenie metod posługiwania się klawiaturą i ekranem. Wprowadzanie tekstów i pojedynczych znaków. Wprowadzanie kodów pojedynczych klawiszy, w tym klawiszy funkcyjnych. Czyszczenie ekranu i ustawianie kolorów. Pozycjonowanie tekstów na ekranie. Redagowanie liczb całkowitych i rzeczywistych. Zapamiętywanie i odtwarzanie fragmentów ekranu.

Proponowane zadania: 1.15–1.17

Ćwiczenie 4. Podstawy grafiki

Cel: Otwieranie i zamykanie trybu graficznego. Wykreślanie linii i figur zależnych od wymiarów ekranu. Wykreślanie i pozycjonowanie tekstu.

Proponowane zadania: 10.15–10.17

Ćwiczenie 5. Definiowanie zmiennych prostych i tablic

Cel: Definiowanie i inicjowanie zmiennych różnych klas pamięci. Definiowanie i inicjowanie tablic. Kompilowanie programu napisanego w dwu plikach z użyciem zmiennej klasy *extern*.

Proponowane zadania: 2.31–2.37

Ćwiczenie 6. Funkcje – podstawy

Cel: Wyrobienie nawyku dzielenia programu na funkcje. Konstruowanie prostych funkcji.

Proponowane zadania: 5.15–5.18

Ćwiczenie 7. Operatory arytmetyczne i logiczne

Cel: Poznanie rzadziej używanych operatorów arytmetycznych i logicznych, jak: dzielenie całkowite, reszta z dzielenia, koniunkcja, alternatywa, indeksacja itp.

Proponowane zadania: 3.30, 3.31

Ćwiczenie 8. Operatory bitowe

Cel: Poznanie operatorów bitowych.

Proponowane zadania: 3.32–3.34

Ćwiczenie 9. Instrukcje sterujące

Cel: Nauka posługiwania się instrukcjami **if**, **switch** oraz instrukcjami pętli

Proponowane zadania: 4.17–4.22

Ćwiczenie 10. Funkcje rekurencyjne

Cel: Praktyka w układaniu algorytmów rekurencyjnych.

Proponowane zadania: 5.21–5.23

Ćwiczenie 11. Wskaźniki – podstawy

Cel: Poznanie prostych operacji na wskaźnikach. Użycie wskaźników do tablic i do funkcji.

Proponowane zadania: 6.28–6.30

Ćwiczenie 12. Wskaźniki – arytmetyka

Cel: Przyswojenie arytmetyki na wskaźnikach.
Proponowane zadanie: 6.31

Ćwiczenie 13. Przetwarzanie tekstów

Cel: Poznanie funkcji rozpoznawania znaków, funkcji kopiowania i łączenia tekstów. Określanie długości tekstów. Wyszukiwanie znaków w tekście. Konwersja liczb na teksty i tekstów na liczby.
Proponowane zadania: 11.6–11.9

Ćwiczenie 14. Wskaźniki – dynamiczny przydział pamięci

Cel: Układanie programów z alokacją tablic jedno- i dwuindeksowych. Zwalnianie przydzielonej pamięci.
Proponowane zadania: 6.32–6.36

Ćwiczenie 15. Funkcje – argumenty i wyniki

Cel: Doskonalenie umiejętności przekazywania danych do funkcji i wyprowadzania wyników. Przekazywanie wskaźników do funkcji. Zastosowanie funkcji o wyniku typu wskazującego. Argumenty funkcji *main*.
Proponowane zadania: 5.19, 5.20, 5.24, 5.25

Ćwiczenie 16. Struktury, unie i ich tablice

Cel: Praktyka w posługiwaniu się strukturami i uniami. Użycie tablic, których elementami są struktury.
Proponowane zadania: 7.22–7.27

Ćwiczenie 17. Strumienie

Cel: Praktyka w posługiwaniu się standardowym wejściem i wyjściem. Wyprowadzanie wyników na drukarkę i do plików dyskowych. Wprowadzanie danych z plików dyskowych. Wprowadzanie i wyprowadzanie zredagowane i niezredagowane.
Proponowane zadania: 8.15, 8.16, 8.19–8.21

Ćwiczenie 18. Obsługa błędów

Cel: Nauka projektowania reakcji programu na sytuacje błędne, jak np. błąd otwarcia pliku, błąd zapisu do pliku, błąd odczytu z pliku, błąd przydziału pamięci itp.
Proponowane zadania: 12.11, 12.12

Ćwiczenie 19. Grafika tekstowa

Cel: Doskonalenie metod posługiwania się ekranem w trybie tekstowym.
Proponowane zadania: 9.9–9.12

Ćwiczenie 20. Grafika pikselowa

Cel: Doskonalenie metod posługiwania się ekranem w trybie graficznym. Rozpoznawanie sterowników. Skalowanie wykresów. Wykreślanie tekstów różnym krojem pisma. Posługiwanie się oknami graficznymi. Zapamiętywanie i powielanie fragmentów ekranu.

Proponowane zadania: 10.16–10.19

Ćwiczenie 21. Inne techniki programowe

Cel: Poznanie funkcji operacji na plikach. Wykorzystanie zegara i daty systemowej. Dostęp do portów. Wykorzystanie przerwań.

Proponowane zadania: 12.11–12.15

14. Literatura

[1] Aitken P., Jones B., *Język C w 21 dni*, Warszawa, Wydawnictwo PLJ, 1994.

[2] Barkakati N., *Borland C++ 4*, Warszawa, Oficyna Wydawnicza READ ME, 1995.

[3] Barteczko K., *Praktyczne wprowadzenie do programowania obiektowego w języku C++*, Warszawa, Lupus, 1993.

[4] Bielecki J., *Borland C++. Programowanie proceduralne*, Warszawa, Wydawnictwo PLJ, 1991.

[5] Bielecki J., *Borland C++. Biblioteki standardowe*, Warszawa, Wydawnictwo PLJ, 1992.

[6] Bulka D., *Efektywne programowanie w C++*, Warszawa, MIKOM, 2001.

[7] *Borland C++ Version 3.1. Programmer's Guide*, Scots Valley, Borland International, 1992.

[8] *Borland C++ Version 3.1. User's Guide*, Scots Valley, Borland International, 1992.

[9] *Borland C++ Version 3.1. Library Reference*, Scots Valley, Borland International, 1992.

[10] Chomicz P., Ulijasz R., *Programowanie w języku C i C++*, Warszawa, Wydawnictwo PLJ, 1992.

[11] Davis S.R., *C++ dla opornych*, Warszawa, READ ME, 1995.

[12] Delannoy C., *Ćwiczenia z języka C*, Warszawa, WNT, 1993.

[13] Drozdek A., Simon D., *Struktury danych w języku C*, Warszawa, WNT, 1996.

[14] Drozdzewicz P., *Programowanie dla Windows w języku C dla początkujących*, Warszawa, LYNX-SFT, 1994.

[15] Holzner S., *Programowanie w Borland C++*, Warszawa, Intersoftland, 1993.

[16] Kernighan B.W., Ritchie D.M., *Język ANSI–C*, Warszawa, WNT, 2001, 2002.

[17] Lippman S. B., *Podstawy języka C++*, Warszawa, WNT, 2001.

[18] Meryk R., *Borland C++*, Warszawa, HELP, 1993.

[19] Perry G., *C – przewodnik dla zupełnych nowicjuszy*, Warszawa, Wydawnictwo PLJ, 1994.

[20] Perry P., Walnum C., *Pierwsze kroki z C*, Warszawa, Intersoftland, 1994.

[21] Plauger P. J., *Biblioteka standardowa C++*, Warszawa, WNT, 1997.

[22] Prata S., *Język C++. Szkoła programowania*, Wrocław, Robomatic, 2003.

[23] Schildt H., *Leksykon C/C++*, Warszawa, Oficyna Wydawnicza LTP, 2002.

[24] Schildt H., *Programowanie C++*, Warszawa, Wydawnictwo RM, 2002.

[25] Sexton C., *Język C. To proste*, Warszawa, Wydawnictwo RM, 2001.

[26] Stec K., *Wybrane elementy języka C*, Gliwice, Wyd. Politechniki Śląskiej, 2001.

[27] Stroustrup B., *Język C++*, Warszawa, WNT, 2000, 2002.

[28] Submit S., *Programowanie w języku C*, Gliwice, Helion, 2003.

[29] Tondo C., *Język ANSI C. Ćwiczenia i rozwiązania*, Warszawa, WNT, 2003.

[30] Vandevoorde D., *Język C++. Ćwiczenia i rozwiązania*, Warszawa, WNT, 2001.

[31] Zalewski A., *Programowanie w językach C i C++ z wykorzystaniem pakietu Borland C++*, Poznań, NAKOM, 2001.

W książce opisano jeden z bardziej popularnych języków programowania komputerów – język C. Przedstawiono zasady kodowania algorytmów i budowy programów w tym języku oraz ważniejsze funkcje biblioteczne kompilatora Borland C++.

Szczególną uwagę zwrócono na wykorzystanie wskaźników, ich arytmetykę, zastosowanie do wskazywania tablic, funkcji i tekstów. Opisano sposoby dynamicznej alokacji tablic jedno- i dwuindeksowych. Przedstawiono ważniejsze zmienne systemowe predefiniowane przez kompilator Borland C++ oraz ważniejsze instrukcje preprocesora. Zaprezentowano przykłady opracowane i uruchomione w środowisku Borland C++. Treść wzbogacono licznymi zadaniami teoretycznymi i praktycznymi przewidzianymi do realizacji w laboratorium komputerowym.

Książka przeznaczona jest dla osób, które zetknęły się z programowaniem w języku Pascal. W szczególności jest ona napisana dla studentów kierunku informatyka oraz dla słuchaczy policealnych szkół zawodowych kształcących w zawodzie technik informatyk.

Wydawnictwa Politechniki Wrocławskiej
są do nabycia w następujących księgarniach:
"Politechnika"
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław
bud. A-1 PWr., tel. (0-71) 320-25-34
"Tech"
plac Grunwaldzki 13, 50-377 Wrocław
bud. D-1 PWr., tel. (0-71) 320-32-52
Prowadzimy sprzedaż wysyłkową

ISBN