sam na sam z językiem C



biel-jaj

WYDAWNICTWA KOMUNIKACJI I ŁĄCZNOŚCI

Jan Bielecki

ZX SPECTRUM

sam na sam z językiem C

Lechowi, Teresie i Kindze

Wydawnictwa Komunikacji i Łączności
Warszawa 1988

Opiniodawca: dr inż. *Jędrzej Wróblewski*
Okładkę projektował: *Lucjan Madziar*
Redaktor: mgr inż. *Elżbieta Gawin*
Redaktor techniczny: *Jadwiga Majewska*
Korekta: *Teresa Kruszona*

681.3.06

Książka zawiera opis obszernego podzbioru języka **C** i jest adresowana do tych użytkowników mikrokomputera ZX Spectrum (oraz naśladującego go mikrokomputera szkolnego Elwro 800 Jr), którzy posługując się łatwo dostępnym w kraju kompilatorem języka **C** firmy **HiSoft**, chcieliby poznać zasady programowania w tym języku – *najważniejszym, współczesnym języku programowania mikrokomputerów*. Należy oczekiwać, że również i ci, którzy nie mają dostępu do mikrokomputera ZX Spectrum, uznają niniejszą książkę za przydatną lekturę przygotowującą ich do profesjonalnego programowania w języku **C**.



ISBN 83-206-0796-5

© Copyright
by Wydawnictwa Komunikacji i Łączności
Warszawa 1988

SPIS TREŚCI

Przedmowa 4

1. Wprowadzenie 5
2. Struktura programu 10
3. Deklarowanie zmiennych, funkcji i typów 15
4. Konstruowanie i wyznaczanie wartości wyrażeń 31
5. Wykonywanie instrukcji 40
6. Wywoływanie funkcji 49
7. Wstępne przetwarzanie programu 53
8. Posługiwanie się bibliotekami funkcji 57
9. Wykonywanie operacji wejścia/wyjścia 60
10. Rozszerzenia i przykłady 69

Dodatek A – Znaki kodu ASCII 78

Dodatek B – Edytor systemu HiSoft 79

Dodatek C – Wybrane funkcje biblioteczne 86

Dodatek D – Kody błędów 92

Dodatek E – Mikrokomputer Elwro 800 Jr 95

PRZEDMOWA

Jednym z najpopularniejszych, a z pewnością najmodniejszym współczesnym językiem programowania jest **C**. Chociaż pokutuje przekonanie, że jest to język programowania mikrokomputerów 16-bitowych, niejako z założenia wyposażonych w obszerną pamięć operacyjną i zewnętrzną, przyswojenie zasad programowania w tym języku oraz posłużenie się nim do efektywnego programowania zagadnień systemowych, zazwyczaj wymagających użycia asemblera, jest możliwe także w przypadku tak prostego mikrokomputera jak ZX Spectrum.

Ten stan rzeczy wynika ze zrealizowania przez firmę **HiSoft** bardzo udanego kompilatora języka **C**, który wsparty opisem implementacji jest doskonałą pomocą w samodzielnym nauczeniu się języka i nabraniu umiejętności uruchamiania w nim całkiem pokaźnych programów systemowych, w tym m.in. gier.

Niniejsza książka jest adresowana do tych użytkowników ZX Spectrum oraz Elwro 800 Jr (por. Dodatek E), których przygotowanie informatyczne niewiele wykracza poza umiejętność posługiwania się językiem **Basic**, a którzy nie mając dostępu do komputerów profesjonalnych chcieliby poznać jedno z ich najważniejszych narzędzi. Chociaż w zamyśle autora niniejsza książka jest kompletna, należy zachęcić początkujących Czytelników do chociażby pobieżnego zapoznania się z przetłumaczoną na język polski książką Briana Kernighana i Dennisa Ritchiego pt. *Język C*, a po nabraniu biegłości w programowaniu na ZX Spectrum do zapoznania się z moimi książkami pt. *Wprowadzenie do języka C* i *Język C – interpretacja standardu*.

Jan Bielecki

1

WPROWADZENIE

Złączony w jedną całość zestaw programów opracowanych przez firmę HiSoft, umożliwiających przygotowywanie, kompilowanie i wykonywanie programów napisanych w języku **C**, będzie tu nazywany systemem **HiSoft C** albo krótko systemem **HiSoft.**

Załadowanie systemu HiSoft do pamięci komputera wymaga posłużenia się dyrektywą

```
LOAD ""
```

zakończoną naciśnięciem klawisza ENTER. Bezpośrednio po wykonaniu tej czynności należy włączyć odtwarzanie z magnetofonu i wyłączyć je gdy system zostanie załadowany, a na ekranie pojawi się zapytanie

```
Save to Microdrive (y/n)?
```

Jeśli na wymienione zapytanie zostanie udzielona odpowiedź y, to kopia systemu wczytanego do pamięci komputera zostanie umieszczona w pamięci zewnętrznej stacji Microdrive. Ponieważ tylko niewielka liczba komputerów ZX Spectrum jest wyposażona w taką stację, zaleca się udzielenie odpowiedzi n. Niezwłocznie po tym na ekranie pojawi się napis

```
HISOFT C Compiler V1.1
Copyright C 1984 HISOFT
```

a system HiSoft znajdzie się w stanie gotowości do kompilowania pierwszego programu.

Jeśli w takim stanie zostanie wprowadzony z klawiatury dowolny program, np.

```
main()
{
   printf("Pierwszy program");
}
```

to będzie on podlegać natychmiastowemu tłumaczeniu na język wewnętrzny komputera.

Samo wprowadzanie programu nie powinno sprawić specjalnej trudności. Jego zasady są takie, że *naciśnięcie klawisza oznaczonego literą* powoduje *wprowadzenie małej litery*, a *naciśnięcie takiego klawisza jednocześnie z naciśnięciem klawisza* CAPS-SHIFT powoduje *wprowadzenie dużej litery. Wprowadzenie znaku specjalnego*, takiego jak np. nawias okrągły, kwadratowy czy klamrowy wymaga *jednoczesnego naciśnięcia klawisza* SYMBOL-SHIFT i *klawisza* oznaczonego *dodatkowo danym znakiem specjalnym*. W szczególności wprowadzenie nawiasów okrągłych wymaga takiego naciśnięcia klawiszy oznaczonych cyframi *8* i *9*, a wprowadzenie nawiasów klamrowych wymaga analogicznego naciśnięcia klawiszy oznaczonych literami *F* i *G*. Jeśli podczas wprowadzania wiersza zostanie naciśnięty niewłaściwy klawisz, to pomyłkę można usunąć, posługując się klawiszem DELETE (CAPS-SHIFT 0).

Po wprowadzeniu tekstu programu należy przekazać systemowi informację o zakończeniu wprowadzania programu źródłowego. Odbywa się to za pomocą jednoczesnego naciśnięcia klawisza EOF (SYMBOL-SHIFT I). W tym momencie zostanie zakończone kompilowanie programu, a na ekranie pojawi się napis

Type y to run:

informujący, że wprowadzenie z klawiatury znaku y spowoduje wykonanie właśnie skompilowanego programu. Jeśli taka czynność istotnie zostanie wykonana, to na ekranie pojawi się napis

Pierwszy program

stanowiący rezultat wykonania pierwszego programu napisanego w języku **C**.

Po wyprowadzeniu tego napisu na ekranie ponownie pojawi się zachęta

Type y to run:

a w odpowiedzi na każde wprowadzenie z klawiatury znaku y nastąpi ponowne wykonanie programu. Jeśli zostanie udzielona jakakolwiek inna odpowiedź, np. n, to system HiSoft przejdzie do stanu początkowego, w którym będzie gotowy do kompilowania nowego programu.

Przytoczony sposób postępowania, tj. przekazywanie programu bezpośrednio do kompilowania, jest stosowany bardzo rzadko i dotyczy zazwyczaj tylko programów bardzo prostych. Znacznie częściej program jest *wstępnie przygotowywany* za pomocą *edytora*, a dopiero *później kompilowany*. Umożliwia to poprawianie za pomocą edytora kolejnych wersji uruchamianego programu.

Jeśli system HiSoft znajduje się w stanie kompilowania, to w celu wywołania edytora należy wprowadzić z klawiatury znak EDIT (CAPS-SHIFT 1), a bezpośrednio po nim znak ENTER. Spowoduje to zgłoszenie się edytora i pojawienie na ekranie zachęty do wprowadzania dyrektywy mającej postać znaku >*(większe)*. W tym momencie można przystąpić do przygotowania programu źródłowego. Odbywa się to w sposób zbliżony do przygotowywania programów w języku **Basic.**

Każdy wiersz programu wprowadzonego w stanie edycji musi być poprzedzony liczbą całkowitą bez znaku. Kolejność wprowadzania wierszy jest nieistotna. Edytor uporządkuje je według opatrujących je liczb. W szczególności, jeśli zostaną wprowadzone wiersze

```
30    printf("Drugi program");
10 main()
40 }
20 {
```

to zostaną uporządkowane w program

```
10 main()
20 {
30    printf("Drugi program");
40 }
```

W celu zastąpienia wiersza programu innym wystarczy ponownie wprowadzić wiersz opatrzony daną liczbą, a w celu usunięcia wiersza opatrzonego pewną liczbą wystarczy wprowadzić z klawiatury wiersz składający się z tej właśnie liczby. Podczas wprowadzania znaków wiersza można eliminować błędnie wprowadzone znaki naciskając klawisz DELETE (CAPS-SHIFT 0). Oczywiście wprowadzenie każdego wiersza musi być kończone naciśnięciem klawisza ENTER.

W każdej chwili przygotowywania programu można uzyskać jego wydruk na ekranie. W tym celu należy posłużyć się dyrektywą L zakończoną naciśnięciem klawisza ENTER. Jej wykonanie spowoduje wyprowadzenie na ekran monitora co najwyżej 10 wierszy. Następne wiersze programu, o ile istnieją, będą wyprowadzane po kolejnych naciśnięciach znaku spacji. Przerwanie wyprowadzania nastąpi po ujawnieniu całego programu albo po naciśnięciu klawisza EDIT (CAPS-SHIFT 1).

Bezpośrednio po przygotowaniu programu źródłowego za pomocą edytora, można przystąpić do jego skompilowania. W tym celu należy wywołać kompilator posługując się dyrektywą C także zakończoną naciśnięciem klawisza ENTER. Po zgłoszeniu się kompilatora należy posłużyć się specjalną dyrektywą

```
#include
```

Jej zinterpretowanie spowoduje włączenie w miejscu jej wystąpienia całego programu przygotowanego przez edytor i natychmiastowe skompi-

lowanie go. Jeśli program jest kompletny, to po wykonaniu tych czynności należy nacisnąć klawisz EOF (SYMBOL-SHIFT I). Spowoduje to wyprowadzenie znanej zachęty

Type y to run:

po której może nastąpić wykonanie programu. Jeśli zostanie naciśnięty klawisz y, to program zostanie wykonany i na ekran monitora zostanie wyprowadzony napis

Drugi program

Jeśli w odpowiedzi na ponowne zapytanie, czy program ma zostać wykonany, zostanie udzielona odpowiedź n, system HiSoft znajdzie się w stanie początkowym kompilacji. Można wtedy skompilować nowy program albo ponownie wywołać edytor w celu poprawienia albo przygotowania nowego programu.

Jeśli po wywołaniu edytora za pomocą klawisza EDIT (CAPS-SHIFT 1), zostanie wykonana dyrektywa L, to na ekranie pojawi się tekst uprzednio wykonywanego programu źródłowego. Wynika to stąd, że *kompilacja i wykonanie programu nie naruszają programu źródłowego,* stale przechowywanego w pamięci operacyjnej.

Jeśli w tym momencie jest pożądane przygotowanie *nowego* programu, to najpierw należy usunąć program *poprzedni.* Odbywa się to za pomocą dyrektywy

Dm,n

w której m oraz n są liczbami bez znaku. Wykonanie takiej dyrektywy, zakończonej jak zwykle naciśnięciem klawisza ENTER, powoduje usunięcie wierszy opatrzonych liczbami z przedziału m...n. W szczególności wykonanie dyrektywy

D1,32767

powoduje usunięcie wszystkich wierszy programu.

Wprowadzanie wierszy programu i opatrywanie ich liczbami może być znacznie ułatwione za pomocą dyrektywy

Im,n

Wykonanie takiej dyrektywy powoduje automatyczne generowanie numerów wierszy począwszy od m i z krokiem n. Każde naciśnięcie klawisza ENTER powoduje zakończenie kompletowania bieżącego wiersza, wygenerowanie numeru wiersza następnego i przystąpienie do kompletowania tego wiersza. Wykonywanie dyrektywy Im,n kończy się w chwili, gdy po wygenerowaniu kolejnego numeru wiersza zostanie naciśnięty klawisz EDIT (CAPS-SHIFT 1).

W szczególności, jeśli zostanie wykonana dyrektywa

I200,10

a po niej zostanie wprowadzony ciąg wierszy

```
main()
{
   printf(Trzeci program");
}
```

zakończony naciśnięciem klawisza EDIT (CAPS-SHIFT 1), to po wykonaniu dyrektywy L na ekranie pojawi się napis

```
200 main()
210 {
220 printf(Trzeci program");
230 }
```

Jeśli obecnie zostanie wywołany kompilator (dyrektywa C), a następnie skompilowany tekst źródłowy przygotowany za pomocą edytora (dyrektywa #include), to zanim wyłoni się sposobność zakomunikowania kompilatorowi, że program jest kompletny na ekranie pojawi się komunikat

ERROR 37

undefined variable

informujący o wykryciu *błędu* w programie.

W Dodatku *D* można znaleźć następujące tłumaczenie tego komunikatu

ERROR – 37 – undefined variable
odwołano się do zmiennej nie zadeklarowanej

W istocie błąd polega nie na tym, że odwołano się do nie zadeklarowanej zmiennej Trzeci, lecz na tym, iż przed literą T zabrakło znaku " *(cudzysłów),* ale tego żaden kompilator się nie domyśli.

Wywołanie edytora w celu poprawienia błędu może odbywać się na kilka sposobów. Wszystkie one oraz zasady posługiwania się wieloma nie wspomnianymi tu możliwościami edytora przedstawiono w Dodatku *B*. Dokładne zapoznanie się z tym dodatkiem znakomicie ułatwi efektywne przygotowywanie i poprawianie programów.

W tym miejscu wystarczy zalecić, aby po pojawieniu się komunikatu o błędzie został naciśnięty klawisz EDIT (CAPS-SHIFT 1), a bezpośrednio po nim znak **Q**. Spowoduje to wywołanie edytora i wyprowadzenie na ekran wiersza, w którym został zidentyfikowany błąd. W obecnej sytuacji wystarczy np. wprowadzić z klawiatury wiersz

220 printf("Trzeci program");

a następnie wywołać kompilator i ponowić kompilację już bezbłędnego programu.

2
STRUKTURA PROGRAMU

Program napisany w języku **C** *składa się z definicji funkcji oraz z deklaracji funkcji i zmiennych. Każdy program musi zawierać definicję funkcji o nazwie main.* Wykonywanie programu rozpoczyna się od tej właśnie funkcji.

Najkrótszy program ma postać

```
main()
{
}
```

W programie tym main jest nazwą funkcji, nawiasy okrągłe obejmują pustą listę parametrów funkcji, a para nawiasów klamrowych stanowiących ciało funkcji nie zawiera ani jednej instrukcji. Wykonanie takiego programu nie wywołuje oczywiście żadnych skutków.

Przykładem nieco bardziej złożonego programu jest

```
int Number = 44;
main()
{
    printf("%d", Number);
}
```

Program ten składa się z deklaracji zmiennej Number oraz z definicji funkcji main. Number jest zadeklarowane jako zmienna całkowita typu (int). W deklaracji przypisano jej daną początkową o wartości 44. Ciało funkcji main jest instrukcją grupującą zawierającą wywołanie funkcji printf. Ponieważ printf jest funkcją wbudowaną w system HiSoft, zbędne jest przytoczenie jej definicji.

Funkcja printf należy do kategorii funkcji wejścia/wyjścia. W odróżnieniu od większości funkcji języka C może być ona wywoływana ze zmienną liczbą argumentów. Pierwszy argument funkcji printf repre-

zentuje wówczas ciąg znaków, który określa sposób wyprowadzenia pozostałych argumentów.

Szczegółowy opis funkcji printf zostanie podany w jednym z dalszych rozdziałów. W tym miejscu wystarczy wyjaśnić, że ciąg znaków reprezentowany przez pierwszy argument tej funkcji może w ogólnym przypadku składać się z wzorców wyprowadzania, opisów znaków i znaków widocznych. Spośród wzorców wyprowadzania najważniejszymi są: %d, %c i %s. Zinterpretowanie wzorca %d powoduje wyprowadzenie argumentu jako liczby dziesiętnej, zinterpretowanie wzorca %c powoduje wyprowadzenie argumentu jako znaku, a zinterpretowanie wzorca %s powoduje wyprowadzenie ciągu znaków wskazywanego przez argument. Zinterpretowanie opisu znaku, którym może być m.in. \n *(nowy wiersz)*, \r *(powrót karetki)* i \t (tabulacja) powoduje odpowiednio: przemieszczenie kursora na początek nowego wiersza, przemieszczenie go na początek bieżącego wiersza i przemieszczenie go na najbliższą pozycję tabulacyjną. Pozostaje wyjaśnić, że zinterpretowanie znaku nie wchodzącego w skład wzorca ani opisu znaku, powoduje wyprowadzenie tego właśnie znaku.

W myśl takich ustaleń wykonanie rozpatrywanego programu powoduje wyprowadzenie wartości zmiennej Number zgodnie z wzorcem %d, tzn. wyprowadzenie liczby 44. Gdyby natomiast instrukcję printf zastąpiono instrukcją

```
printf("Jan%c is %d %s", 'B',44,"now");
```

to nastąpiłoby wyprowadzenie napisu

```
JanB is 44 now
```

Rezultat taki miałby następujące uzasadnienie: zinterpretowanie liter J, a i n spowodowałoby wyprowadzenie napisu Jan; zinterpretowanie wzorca %c spowodowałoby wyprowadzenie argumentu 'B' jako litery B; zinterpretowanie liter is i otaczających je spacji spowodowałoby wyprowadzenie napisu is; zinterpretowanie wzorca %d spowodowałoby wyprowadzenie argumentu 44 jako liczby; zinterpretowanie wzorca %s spowodowałoby wyprowadzenie ciągu znaków reprezentowanego przez literał "now". Gdyby natomiast chodziło o uzyskanie napisu o postaci

```
JanB
is
44
now
```

to należałoby posłużyć się instrukcją

```
printf("Jan%c \nis \n%d \n%s", 'B',44,"now");
```

W instrukcji takiej wystąpiłby trzykrotnie opis znaku nowego wiersza \n. Każde jego zinterpretowanie powodowałoby zakończenie komple-

towania bieżącego wiersza i przystąpienie do kompletowania wiersza następnego.

Poprzestając na tym krótkim opisie zasad posługiwania się funkcją printf pozostaje nadmienić, że zakończenie wykonywania tej funkcji nie powoduje niejawnego zakończenia kompletowania bieżącego wiersza. Z tego powodu wykonanie np. instrukcji

```
printf("%d %s %d", 13,"grudnia",1981);
```

ma taki sam skutek jak wykonanie sekwencji trzech instrukcji

```
printf("%d", 13);
printf(" %s", "grudnia");
printf(" %d", 1981);
```

Wracając do omawiania struktury programów można przytoczyć program

```
main()
{
   int Number;

   Number = 44;
   output(Number);
}

output(Num)
   int Num;
{
   printf("%d", Num);
}
```

Program ten składa się z dwóch definicji funkcji. Definicja funkcji main składa się z nagłówka wyszczególniającego nazwę funkcji wraz z pustą listą jej parametrów oraz z ciała funkcji. Ciało funkcji main jest instrukcją grupującą zawierającą deklarację zmiennej Number, instrukcję przypisania oraz wywołanie funkcji output. Definicja funkcji output składa się z nagłówka wyszczególniającego nazwę funkcji i jej parametr Num oraz z deklaracji tego parametru. Ciało funkcji output jest instrukcją grupującą zawierającą wywołanie funkcji printf. W chwili wywołania funkcji output następuje skojarzenie z parametrem Num argumentu Number i tym samym przypisanie parametrowi danej o wartości 44. Z tego powodu wykonanie instrukcji printf powoduje wyprowadzenie liczby 44.

Traktowanie programu jako zestawu definicji i deklaracji nie jest jedynym sposobem jego podziału na części. Programy mogą być rozpatrywane również jako zestawy jednostek leksykalnych i komentarzy,

ewentualnie oddzielonych odstępami: znakami spacji, tabulacji i przejścia do nowego wiersza.

Jednostkami leksykalnymi są słowa kluczowe, identyfikatory, literały, operatory i ograniczniki. Podział programu na jednostki leksykalne odbywa się w kolejności występowania znaków w programie, a za kolejną jednostkę leksykalną jest uznawany najdłuższy ciąg znaków, który może stanowić taką jednostkę. Oznacza to w szczególności, że ponieważ operatorami są zarówno – *(minus)* jak i – – *(podwójny minus)*, wyrażenie takie jak np.

a – – – b

jest traktowane jak

a – – – b

a nie jak różne od niego wyrażenie

a – – – b

Komentarzem jest napis rozpoczynający się od symbolu /* *(kreska ukośna, gwiazdka)* i kończący się najbliższym symbolem */ *(gwiazdka, kreska ukośna)*. W komentarzach nie mogą być zawarte inne komentarze. Z punktu widzenia podziału programu na jednostki leksykalne każdy komentarz jest traktowany tak jak odstęp, a więc może spełniać funkcję separatora jednostek leksykalnych.

Słowem kluczowym jest napis składający się z ustalonego ciągu małych liter. Słowa kluczowe są zastrzeżone i mogą być używane tylko w przypisanym im znaczeniu. Pełen ich wykaz obejmuje następujące słowa

auto	else	long	typedef
break	entry	register	union
case	extern	return	unsigned
char	float	short	while
continue	for	sizeof	
default	goto	static	– – – – – – –
do	if	struct	fortran
double	int	switch	asm

Słowa kluczowe nie mogą być używane do oznaczania obiektów programowych takich jak np. funkcje i zmienne.

Uwaga: W systemie HiSoft dodatkowymi słowami kluczowymi są inline i cast.

Identyfikatorami są – różne od słów kluczowych – ciągi literowo-cyfrowe rozpoczynające się od litery. W sensie języka **C** literą jest także znak podkreślenia. Litery małe i duże są uznawane za różne. Jeśli 8 pierwszych znaków dwóch identyfikatorów jest identycznych, to identyfikatory uważa się za nieodróżnialne.

W języku **C** występują trzy rodzaje *literałów:* literały liczbowe *(liczby),* literały znakowe *(znaki)* i literały łańcuchowe *(łańcuchy).* Literał liczbowy składa się z ciągu cyfr dziesiętnych. Literał znakowy ma postać napisu 'c', w którym c jest dowolnym znakiem widocznym różnym od znaków ' *(apostrof)* i \ *(ukośnik)* albo opisem znaku. Najczęściej używanymi opisami znaków są: \0 (opis znaku o kodzie 0), \n *(opis znaku nowego wiersza),* \r *(opis znaku powrotu karetki),* \t *(opis znaku tabulacji),* \' *(opis znaku apostrof)* i \\ *(opis znaku ukośnika).* Literał łańcuchowy ma postać napisu "s", w którym s jest dowolnym ciągiem znaków oraz opisów znaków. Jeśli w literale ma być reprezentowany znak " *(cudzysłów),* to jest zapisywany za pomocą opisu \" *(ukośnik, cudzysłów).* Każdy literał łańcuchowy reprezentuje stałą wskazującą. Stała ta wskazuje ciąg znaków wymienionych między parą apostrofów, uzupełniony dodatkowym znakiem o kodzie 0 stanowiącym ogranicznik tego ciągu.

W odróżnieniu od innych języków programowania, literał znakowy nie stanowi szczególnego przypadku literału łańcuchowego. Wynika to stąd, że napis taki jak np. 'j' reprezentuje jednoznakową daną, która ma wartość równą kodowi znaku j, natomiast napis taki jak np. "j" reprezentuje daną wskazującą pierwszy znak z ciągu znaków składającego się ze znaku j oraz ze znaku o kodzie 0. To właśnie stanowi niejawne rozszerzenie ciągu znaków o znak o kodzie 0, stanowi m.in. o różnicy rozpatrywanych literałów.

Biorąc pod uwagę podane definicje i wyjaśnienia, można zauważyć, że w programie

```
int Age = 44; /* Dla roku 1987 */
main()
{
   printf("Jan%c is %d \"%s \"", 'B',Age, "now");
}
```

występują m.in.: komentarz /* Dla roku 1987 */, słowo kluczowe int, identyfikatory Age, main i printf, literał znakowy 'B', literał łańcuchowy "now", operator = *(równa się)* oraz takie ograniczniki jak np. , *(przecinek),* nawiasy okrągłe i ; *(średnik).* Wykonanie przytoczonego programu powoduje wyprowadzenie napisu

```
JanB is 44 "now"
```

3

DEKLAROWANIE ZMIENNYCH, FUNKCJI I TYPÓW

Przedmiotem przetwarzania opisanego za pomocą programu są dane. Dane występują w programie pod postacią stałych i zmiennych i są reprezentowane przez nazwy: nazwy stałych i nazwy zmiennych. Najprostszymi nazwami stałych są literały, a najprostszymi nazwami zmiennych są identyfikatory. W ogólnym przypadku dane mogą być reprezentowane przez wyrażenia. Z tego powodu odpowiednio skonstruowane wyrażenia będą także uznawane za nazwy danych.

Właściwości danych reprezentowanych przez literały wynikają z samego zapisu literałów, a zatem deklarowanie literałów jest zbyteczne. Inaczej jest ze zmiennymi. Zmienne muszą być zawsze deklarowane, a deklaracja zmiennej musi wystąpić przed jej użyciem w programie.

Zmienne programu można podzielić na proste i złożone. Zmiennymi prostymi są te zmienne, którym można przypisać jedynie dane skalarne (niepodzielne). Zmienne złożone są natomiast agregatami danych: tablicami, strukturami i uniami.

Deklaracja zmiennych ma na ogół postać

oznaczenie-typu lista-deklaratorów ;

np.

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

z tym, że elementy listy-deklaratorów nie muszą być identyfikatorami, lecz mogą mieć postaci bardziej złożone, jak np. w deklaracji

```
int *(*(*fun())[2])();
```

Zasady konstruowania deklaratorów są następujące: jeśli pewien napis Dec jest deklaratorem, to deklaratorem jest także napis (Dec) oznaczający to samo co Dec, napis Dec[w] oznaczający tablicę o w elementach, napis *Dec oznaczający wskazanie obiektu i napis Dec() oznaczający funkcję. W obrębie deklaratora najwyższy priorytet mają nawiasy okrągłe obejmujące deklarator, a najniższy ma znak * *(gwiazdka)*. Anali-

zę złożonego deklaratora należy zaczynać od identyfikatora i kontynuować ją (z uwzględnieniem priorytetów) od wnętrza ku zewnętrzu deklaracji. Należy przy tym pamiętać, że jeśli pewien deklarator opisuje funkcję, to jest poprawny jedynie wtedy, gdy rezultat funkcji jest daną skalarną, a więc nie np. funkcją, albo tablicą.

Biorąc pod uwagę przytoczone zasady ogólne, łatwo jest konstruować i analizować złożone deklaratory. W szczególności można stwierdzić, że np. deklaracja

```
int (*Name())[2];
```

równoważna po uwzględnieniu priorytetów deklaracji

```
int (*(Name()))[2];
```

analizowana "od wnętrza ku zewnętrzu" jest deklaracją funkcji Name, której rezultatem jest wskazanie dwuelementowej tablicy danych typu (int).

W odróżnieniu od rozpatrzonej deklaracji poprawnej, deklaracją błędną jest natomiast deklaracja

```
char Fun()[3];
```

równoważna deklaracji

```
char (Fun())[3];
```

a to dlatego, że deklaruje ona funkcję, której rezultatem jest 3-elementowa tablica danych typu (char), a więc obiekt, który nie jest skalarem.

Pozostaje nadmienić, że jeśli pewien obiekt programu został zadeklarowany za pomocą deklaracji

```
oznaczenie-typu deklarator
```

to przyjmuje się, że jest on typu

```
(oznaczenie-typu pseudodeklarator)
```

w którym to napisie pseudodeklarator różni się tylko tym od deklaratora, że nie zawiera identyfikatora obiektu. W tym sensie np. zmienna Ptr zadeklarowana jako

```
int (*Ptr)[2][3];
```

jest typu (int (*)[2][3]).

Deklarowanie zmiennych prostych

W rozpatrywanym podzbiorze języka występują następujące rodzaje *zmiennych prostych*

- zmienne typu *(char),* którym można przypisywać dane całkowite z przedziału 0...255,

• zmienne typu *(int)*, którym można przypisywać dane całkowite z przedziału – 32768...32767,

• zmienne typu *(unsigned)*, którym można przypisywać dane całkowite z przedziału 0...65535,

• zmienne wskazujące, którym można przypisywać wskazania zmiennych.

Deklarowanie zmiennych może być połączone z przypisywaniem im danych początkowych. W takim przypadku wymaga się jedynie, aby dana początkowa była określona przez *wyrażenie stałe*, tj. wyrażenie, którego wartość może zostać wyznaczona jeszcze przed podjęciem wykonywania programu. Przyjmuje się ponadto, że jeśli deklaracja zmiennej wystąpiła na zewnątrz definicji funkcji, a w deklaracji nie wystąpiło przypisanie danej, to deklaracja jest traktowana tak, jakby zawierała przypisanie danej o wartości 0.

Uwaga: W rozpatrywanym podzbiorze implementacji HiSoft, przypisania danych mogą wystąpić jedynie w deklaracjach występujących na zewnątrz definicji funkcji.

W przykładowym programie

```
char Initial = 'J',
     Name = 'B';
int Age = 44;

main()

{
   int Year;

   Year = 1987;
   printf("%c%c is %d in %d", Initial,
                              Name, Age,
                              Year);
}
```

zadeklarowano zmienne Initial i Name typu (char) oraz zmienne Age i Year typu (int). Przypisanie danej początkowej reprezentowanej przez literał 1987 zrealizowano za pomocą odrębnej instrukcji. Wykonanie programu powoduje wyprowadzenie napisu

```
JB is 44 in 1987
```

W nawiązaniu do poprzednich ustaleń należy nadmienić, że zmiennej Name przypisano daną całkowitą typu (char) o wartości równej kodowi litery B, natomiast zmiennej Age przypisano daną całkowitą typu (int) o wartości 44. Gdyby deklaracja zmiennej Age nie zawierała przypisania danej początkowej, a więc gdyby miała postać

```
int Age;
```

to przez domniemanie byłaby traktowana tak jak deklaracja

```
int Age = 0;
```

Analogiczne domniemanie nie dotyczy zmiennej Year, ponieważ nie została ona zadeklarowana na zewnątrz definicji funkcji.

Zanim zostanie przytoczony program ilustrujący użycie zmiennych wskazujących, należy wyjaśnić co to jest *wskazanie*. Jako definicję można przyjąć, że wskazanie jest daną określającą położenie innej danej. Chociaż wielu programistów utożsamia pojęcie wskazania z adresem, słowo adres nie będzie tu używane, ponieważ jest ono obce językowi **C**, a *traktowanie wskazań jako adresów może stać się źródłem trudnych do wykrycia błędów programowania.*

W celu przekształcenia wyrażenia reprezentującego pewną daną, w wyrażenie reprezentujące wskazanie tej danej należy posłużyć się *operatorem wskazania & (ampersand)*. W tym celu wystarczy rozpatrywane wyrażenie poprzedzić tym operatorem.

Operatorem odwrotnym do operatora wskazania jest *operator wyłuskania * (gwiazdka)*. Jeśli pewne wyrażenie reprezentujące wskazanie danej zostanie poprzedzone operatorem wyłuskania, to staje się wyrażeniem reprezentującym wskazywaną daną.

Z definicji przyjmuje się, że literał łańcuchowy reprezentuje daną wskazującą pierwszy znak ciągu znaków określonego przez ten literał. Z tego powodu literał taki jak np. "jb" reprezentuje daną wskazującą znak j, a wyrażenie *"jb" reprezentuje znak j ciągu znaków składającego się ze znaku j, znaku b i znaku o kodzie 0.

Należy nadmienić, że o ile operator wyłuskania może być użyty w odniesieniu do dowolnego wyrażenia wskazującego (tzn. reprezentującego wskazania danej), o tyle operator wskazania może być użyty tylko w odniesieniu do wyrażenia, które reprezentuje zmienną prostą (a nie np. wskazanie takiej zmiennej). Wynika stąd w szczególności, że mimo iż operator wskazania jest operatorem odwrotnym do operatora wyłuskania, tożsamość

```
&*exp == exp
```

jest prawdziwa wtedy i tylko wtedy gdy exp jest wyrażeniem reprezentującym daną wskazującą zmienną prostą, a nie np. stałą, jak ma to miejsce w przypadku wyrażenia "jb".

Chociaż w większości języków programowania, w których występują wyrażenia wskazujące, jedynymi dopuszczalnymi na nich operacjami są wyłuskania, przypisania i porównania, w języku **C** możliwe jest dodawanie lub odejmowanie od wyrażenia wskazującego innego wyrażenia reprezentującego daną całkowitą. W takim przypadku przyjmuje się, że jeśli Ptr jest wyrażeniem wskazującym i-ty element pewnego ciągu przyległych danych, to wyrażenie

```
Ptr + Num
```

reprezentuje wskazanie elementu zajmującego pozycję i +Num. Ozna-

cza to w szczególności, że jeśli Ptr reprezentuje wskazanie pewnego znaku należącego do spójnego ciągu znaków, to Ptr + 1 reprezentuje wskazanie znaku następnego.

Ponieważ operacje dodawania do wyrażeń wskazujących wyrażeń całkowitych oraz stosowanie wyłuskiwania występują w języku bardzo często, przyjęto z definicji, że każdy poprawny napis o postaci

```
*(Ptr + Num)
```

jest równoważny napisowi

```
Ptr[Num]
```

oraz napisowi

```
Num[Ptr]
```

Występująca w tych napisach para nawiasów kwadratowych będzie dalej nazywana *operatorem indeksowania.*

Korzystając z przytoczonego opisu i definicji można rozpatrzyć następujący program

```
char *Ref = "Jan";

main()
{
   printf("%s = %c%c%c", Ref,*Ref,
                         *(Ref + 1),
                         Ref[2]);
}
```

W programie tym Ref zadeklarowano jako zmienną, której można przypisywać wskazania danych typu (char). Tym samym zmienna reprezentowana w programie przez identyfikator Ref jest typu (char *). Podczas deklarowania, zmiennej Ref przypisano daną wskazującą literę J czteroznakowego ciągu znaków składającego się ze znaków J, a, n i \0. Podczas wykonywania instrukcji, w której występuje wywołanie funkcji printf następuje skojarzenie wzorca %s ze wskazaniem przypisanym zmiennej Ref. Powoduje to wyprowadzenie ciągu znaków rozpoczynającego się od wskazanego miejsca aż do znaku o kodzie 0 wyłącznie, a więc wyprowadzenie napisu Jan. Jeśli Ref reprezentuje wskazanie litery J, to *Ref reprezentuje literę J. Dlatego skojarzenie pierwszego wzorca %c z argumentem *Ref spowoduje wyprowadzenie litery J. Ponieważ Ref reprezentuje wskazanie litery J, więc Ref + 1 reprezentuje wskazanie litery a, a więc *(Ref + 1) reprezentuje literę a, a zatem zinterpretowanie drugiego wzorca %c spowoduje wyprowadzenie litery a. Wyrażenie Ref[2] jest uproszczonym przedstawieniem wyrażenia *(Ref + 2), a zatem reprezentuje literę n. Tak więc ostatecznie, wykonanie programu spowoduje wyprowadzenie napisu

Jan = Jan

Nawiązując do rozpatrzonych dotąd przykładów można podać, że

```
int Num;     jest deklaracją zmiennej typu (int)
int *Ptr;    jest deklaracją zmiennej typu (int *)
int **Ref;   jest deklaracją zmiennej typu (int **)
```

Zmiennej Num mogą być przypisywane dane typu (int). Zmiennej Ptr mogą być przypisywane dane typu (int *), a więc dane wskazujące dane typu (int). Zmiennej Ref mogą być przypisywane dane typu (int **), a więc dane wskazujące dane, które wskazują dane typu (int). Przypadki, gdy w programie należy posłużyć się więcej niż dwoma poziomami wskazania pośredniego są bardzo rzadkie.

Deklarowanie tablic

Tablica jest zmienną złożoną, której wszystkie komponenty są takiego samego typu. Komponenty te będą nazywane elementami tablic.

Deklaracja tablicy różni się tym od deklaracji zmiennej prostej, że występuje w niej, ujęte w nawiasy kwadratowe, wyrażenie stałe określające liczbę elementów tablicy. Mimo iż tablice występujące w języku **C** są w istocie jednowymiarowe, istnieje możliwość deklarowania tablic których elementami są tablice, a więc tym samym deklarowania tablic wielowymiarowych. Tablicom zadeklarowanym na zewnątrz definicji funkcji można podczas deklarowania przypisywać dane początkowe. Napis określający dane początkowe, nazywany dalej inicjatorem, składa się z listy literałów ujętej w nawiasy klamrowe. Jeśli lista literałów jest niekompletna, to przyjmuje się, że jej brakującymi elementami są literały 0.

W następującym programie zadeklarowano dwuwymiarową tablicę Source i jednowymiarową tablicę Target. Elementom tablicy Source przypisano podczas deklarowania dane początkowe.

```
char Source[3][3] = { { 'E','w','a'},
                      {'I','z','a'},
                      {'J','a','n'} };

main()
{
    char Target[4];
    int i;

    for(i=0; i < 4; i++)
      Target[i] = Source[2][i];
    Target[3] = '\0';
    printf("%s", Target);
}
```

Tablica Source składa się z 3 wierszy i z 3 kolumn. Elementy tablicy są uporządkowane wierszami, a każdy z nich jest typu (char). Lewy-górny narożnik tablicy jest elementem Source[0][0], a prawy dolny jest elementem Source[2][2]. Tablica Target składa się z 4 elementów: Target[0], Target[1], Target[2] i Target[3]. Wykonanie instrukcji powtarzającej for powoduje przypisanie danych stanowiących drugi wiersz tablicy Source, pierwszym trzem zmiennym tablicy Target. Podczas wykonywania instrukcji powtarzającej, zmiennej i są przypisywane dane o wartościach 0...3. Zapis instrukcji można odczytać następująco: "dla i = 0 oraz tak długo jak długo jest i < 4, wykonywać przypisanie, a następnie zwiększać i o 1". Po zakończeniu wykonywania instrukcji powtarzającej następuje umieszczenie w tablicy Target dodatkowego znaku o kodzie 0, a następnie wykonanie instrukcji zawierającej wywołanie funkcji printf. Ponieważ w języku przyjęto, że każde wyrażenie reprezentujące tablicę jest natychmiast niejawnie przekształcane w wyrażenie reprezentujące wskazanie pierwszego elementu tej tablicy, instrukcja

```
printf("%s", Target);
```

jest traktowana tak jak instrukcja

```
printf("%s", &Target[0]);
```

a zatem jej wykonanie spowoduje wyprowadzenie napisu Jan. Ten specjalny sposób traktowania nazw tablic tak, jakby były nazwami wskazań powoduje, że wykonywanie operacji na tablicach może być w dowolnym kontekście zastąpione wykonywaniem operacji na danych wskazujących.

Elementami tablic nie muszą być wyłącznie zmienne typu (char), zmienne typu (int) oraz tablice. Mogą być nimi także struktury, unie i zmienne wskazujące. Ten ostatni przypadek ilustruje następujący program.

```
char*Family[3] = { "Ewa","Iza","Jan"};

main()
{
   printf("%c & %c", ** Family,
                     *Family[2]);
}
```

W programie tym Family jest 3-elementową tablicą o elementach typu (char *). Elementami tablicy są więc wskazania danych typu (char). Podczas deklarowania tablicy jej elementowi Family[0] zostaje przypisane wskazanie pierwszego znaku ciągu Ewa, elementowi Family[1] zostaje przypisane wskazanie pierwszego znaku ciągu Iza, a elementowi Family[2] zostaje przypisane wskazanie pierwszego znaku ciągu Jan. Ponie-

waż wyrażenie Family reprezentuje wskazanie elementu Family[0], wyrażenie *Family reprezentuje element Family[0], a więc **Family reprezentuje obiekt wskazywany przez Family[0], tj. znak E. Analogicznie, Family[2] reprezentuje element tablicy Family wskazujący znak J, a więc *Family[2] reprezentuje znak J. Wykonanie programu powoduje zatem wyprowadzenie napisu E & J. Nie od rzeczy będzie odnotować, że o ile sama tablica Family jest typu (char *[3]), tzn. jest trzyelementową tablicą wskazań danych typu (char), o tyle wyrażenie Family jest typu (char **), wyrażenie *Family jest typu (char *), a wyrażenie **Family jest typu (char).

Jako przeciwstawienie programu, w którym występowała tablica wskazań, zostanie obecnie rozpatrzony program, w którym występuje zmienna wskazująca tablicę.

```
int Array[2][2] = { {3,2},{1,0} },
    (*Ptr)[2] = Array + 1;

main()
{
   printf("%d%d", **Ptr,Ptr[-1][0]);
}
```

W programie tym, Array jest dwuelementową tablicą, której elementami są dwuelementowe tablice o elementach typu (int). Tablica Array może być rozpatrywana jako tablica dwuwymiarowa, której elementom przypisano dane o wartościach 3, 2, 1 i 0. Ptr jest natomiast zmienną prostą wskazującą dwuelementowe tablice typu (int). Jest to więc zmienna typu (int (*)[2]). Zmiennej Ptr przypisano w deklaracji daną reprezentowaną przez wyrażenie Array + 1. Ponieważ wyrażenie Array reprezentuje wskazanie tablicy stanowiącej pierwszy wiersz tablicy Array, więc wyrażenie Array + 1 reprezentuje wskazanie tablicy stanowiącej drugi wiersz. To wskazanie zostaje przypisane zmiennej Ptr. Przypisanie jest poprawne, ponieważ zarówno Ptr jak i Array + 1 są tego samego typu (int (*)[2]). Po dokonaniu przypisania, *Ptr reprezentuje tablicę stanowiącą drugi wiersz tablicy Array, a więc w istocie reprezentuje wskazanie zerowego elementu tej tablicy, tj. wskazanie elementu Array[1][0]. Zatem **Ptr reprezentuje ten właśnie element i zinterpretowanie pierwszego wzorca %d powoduje wyprowadzenie liczby 1. Analogicznie, Ptr[-1] reprezentuje tablicę Array[0], a więc Ptr[-1][0] reprezentuje element Array[0][0] i zinterpretowanie drugiego wzorca %d powoduje wyprowadzenie liczby 3. Wykonanie programu powoduje więc wyprowadzenie napisu 13.

Jeśli deklaracja tablicy zawiera przypisanie danych początkowych, to dopuszcza się pominięcie informacji o liczbie elementów tablicy. W przypadku tablicy wielowymiarowej pominięcie to może jednak

dotyczyć tylko pierwszego wymiaru. Określenie liczby elementów tablicy w pierwszym wymiarze zostanie wówczas domniemane na podstawie liczby danych początkowych przypisywanych elementom tablicy. Dozwolone jest również pominięcie nawiasów klamrowych obejmujących kompletne inicjatory wewnętrzne. W takim przypadku rozmieszczenie takich brakujących nawiasów zostanie domniemane na podstawie deklaracji tablicy. Zasady te ilustruje następujący zestaw deklaracji

```
char Vector[] = { 'J','a','n' };
int Array[][2] = { 1,2,3,4,{5},6 };
char *Matrix[][3] = { {"Ewa"}, {"Jan"} };
```

równoważny zestawowi

```
char Vector[3] = {'J','a','n' };
int Array[4][2] = { {1,2}, {3,4},{5,0},{6,0} };
char *Matrix[2][3] = { "Ewa",0,0,"Jan",0,0 };
```

Wyjątkiem od zasady, że napis "s" reprezentuje wskazanie pierwszego z ciągu znaków s, jest ustalenie, że jeśli napis taki wystąpi jako element inicjatora tablicy o elementach typu (char), to poszczególne jego znaki określą dane początkowe przypisane tym elementom. W tym sensie następujący zestaw deklaracji

```
char Vector[] = "Ewa";
char Array[][3] = { "Ewa","Jan" };
```

jest równoważny zestawowi

```
char Vector[4] = { 'E','w','a','\0' };
char Array[2][3] = { { 'E','w','a' } ;
                     { 'J','a','n' } };
```

Deklarowanie struktur i unii

Struktury i *unie* są zmiennymi złożonymi składającymi się z komponentów na ogół różnych typów. Komponenty te będą nazywane *polami*. Ponieważ polami mogą być nie tylko zmienne proste, ale także tablice, struktury i unie, istnieje możliwość posługiwania się strukturami i uniami o dużym stopniu złożoności.

Pola struktur są rozmieszczone w pamięci operacyjnej w kolejności ich wystąpienia w deklaracji struktury. Natomiast każde z pól unii jest rozmieszczone począwszy od tego samego miejsca. Wynika stąd, że jeśli w pewnym programie zostanie zadeklarowana struktura Record

```
struct{
   int Fix;
   char Chr;
   int Arr[3];
   char *Ptr[2];
} Record;
```

o polach Fix, Chr, Arr i Ptr, to zajmie ona tyle miejsca ile łącznie zajmują te pola, natomiast jeśli zostanie zadeklarowana unia Record

```
union{
   int Fix;
   char Chr;
   int Arr[3];
   char *Ptr[2];
} Record;
```

o polach Fix, Chr, Arr i Ptr, to zajmie ona tyle miejsca ile wymaga pole o największym zapotrzebowaniu pamięci.

Biorąc pod uwagę, że w systemie HiSoft każda zmienna typu (char) wymaga jednego bajtu pamięci, a każda zmienna typu (int) oraz każda zmienna wskazująca wymaga dwóch bajtów pamięci, przytoczona struktura zajmie łącznie 11 bajtów pamięci, a przytoczona unia zajmie 6 bajtów pamięci.

Ponieważ różnice między strukturami i uniami ograniczają się jedynie do sposobu rozmieszczenia ich pól oraz do tego, że uniom nie można przypisać danych początkowych, pozostała część opisu będzie dotyczyć jedynie struktur.

Deklaracja struktury składa się na ogół ze słowa kluczowego struct, po którym następuje ujęty w nawiasy klamrowe wykaz pól struktury, a po nim deklarator i znak ; *(średnik)*. Deklarator może się składać z identyfikatora struktury, ale może mieć postaci bardziej złożone, analogiczne do omówionych uprzednio. W miejscu pojedynczego deklaratora może ponadto występować ich lista. W szczególności napis

```
struct{
  int Fix;
  char Chr[2][2];
} Str,Arr[3],*Ptr[4];
```

stanowi: deklarację struktury Str o polach Fix i Chr, deklarację trzyelementowej tablicy struktur Arr oraz deklarację czteroelementowej tablicy wskazań Ptr struktur takich jak Str albo Arr[i].

Jeśli między słowem kluczowym struct i najbliższym nawiasem klamrowym otwierającym zostanie umieszczony identyfikator, to w dalszych deklaracjach może być on użyty do identyfikowania struktury o podanej budowie. W szczególności oznacza to, że deklaracja taka jak np.

```
struct Tag{
  int Fix;
  char Chr;
} One,Two[2];
```

w której Tag jest arbitralnie dobranym identyfikatorem nazywanym dalej *oznacznikiem struktury,* może zostać z równym skutkiem zastąpiona parą deklaracji

```
struct Tag{
  int Fix;
  char Chr;
} One;
struct Tag Two[2];
```

albo parą deklaracji

```
struct Tag{
  int Fix;
  char Chr;
};
struct Tag One,Two[2];
```

Zastosowanie oznacznika struktury jest niezbędne w tych przypadkach, gdy podczas deklarowania pola struktury jest konieczne odwołanie się do typu struktury właśnie deklarowanej, jak ma to miejsce w następującej deklaracji

```
struct List{
  int Data;
  struct List *Link;
} Str;
```

której polu Link mogą być przypisywane wskazania struktur typu (struct List).

Podobnie jak zmiennym prostym, tak i polom struktur (ale nie unii!) mogą być przypisywane dane początkowe. Inicjator struktury składa się w ogólnym przypadku z listy ujętej w nawiasy klamrowe. Elementami listy mogą być wyrażenia stałe, najczęściej literały, albo podlisty. Wyrażenia stałe służą do inicjowania pól skalarnych, a podlisty służą do inicjowania pól, które są tablicami i strukturami. Podlista użyta do zainicjowania pola ma postać inicjatora obiektu o typie tego pola. Jeśli pewna podlista zawiera kompletny zestaw wyrażeń stałych inicjujących dane pole, to obejmujące ją nawiasy klamrowe mogą być pominięte. Jeśli dla pewnych pól nie zostaną określone jawnie dane początkowe, a deklaracja struktury znajduje się na zewnątrz definicji funkcji, to dla takich pól zostaną domniemane literały 0. Należy przypomnieć, że polom struktur zadeklarowanych w obrębie definicji funkcji nie mogą być przypisywane dane początkowe.

W myśl przytoczonego opisu, posłużenie się deklaracją

```
struct{
  char One, Two;
  struct{
    char Uno,Due;
  } InStr;
  char Arr[2],
       *Ref;
  OutStr = { 1,2,{3,4},{5} };
}
```

powoduje przypisanie danych początkowych polom struktury OutStr w taki sposób, że np. polu Arr[0] zostanie przypisana dana o wartości 5, a polu Ref przypisana dana o wartości 0.

Uwaga: W systemie HiSoft, w celu uproszczenia kompilatora, przyjęto założenie, że inicjator struktury nie może zawierać podlist. Powoduje to, że w przytoczonej wyżej deklaracji inicjator powinien zostać zmieniony do postaci

```
{ 1,2,3,4,5 }
```

Istotnym odstępstwem od języka wzorcowego jest ponadto przyjęcie, że przypisywanie danych nie dotyczy poszczególnych pól struktury, lecz że dotyczy obszaru pamięci operacyjnej przydzielonego strukturze. Biorąc to pod uwagę założono, że jeśli elementem listy inicjatora jest wyrażenie o wartości z przedziału 0...255, to następuje zainicjowanie jednego, a w pozostałych przypadkach dwóch kolejnych bajtów obszaru pamięci przydzielonego strukturze. Oznacza to w szczególności, że np. zinterpretowanie deklaracji

```
struct{
  int Uno,Due;
} Str = { 1,1 };
```

powoduje przypisanie polu Uno danej o wartości 257 i polu Due danej o wartości 0, zamiast spowodować przypisanie każdemu z tych pól danej o wartości 1.

Deklarowanie funkcji

Deklaracja *funkcji* składa się z *nagłówka* i z *ciała funkcji*. Nagłówek zawiera nazwę funkcji, ujętą w nawiasy okrągłe, listę nazw parametrów funkcji i deklaracje parametrów. Ciało jest instrukcją grupującą rozpoczynającą się od nawiasu klamrowego otwierającego i kończącą się odpowiadającym mu nawiasem klamrowym zamykającym. Jeśli wykonanie funkcji zakończy się wykonaniem instrukcji

```
return exp;
```

w której exp jest wyrażeniem, to w miejscu wywołania funkcji zostanie

udostępniona dana stanowiąca rezultat funkcji. Typ tej danej można określić na podstawie analizy nagłówka funkcji. W następującym programie, którego wykonanie powoduje wyprowadzenie napisu Bielecki

```
main()
{
    char *Surname();

    printf("%s", Surname("Jan Bielecki",4));
}

char *
Surname(String, Number)
    char *String;
    int Number;
{
    return String + Number;
}
```

występują definicje dwóch funkcji: bezparametrowej funkcji main i dwuparametrowej funkcji Surname. Parametrami funkcji Surname są String i Number. Parametr String jest typu (char *) i zostanie skojarzony z argumentem "Jan Bielecki". Parametr Number jest typu (int) i zostanie skojarzony z argumentem Number. Rezultatem funkcji jest dana reprezentowana przez wyrażenie występujące w instrukcji return. Dana ta jest typu (char *), zgodnego z typem zadeklarowanym w nagłówku funkcji przed nazwą Surname. Ponieważ wywołanie funkcji występuje przed jej zdefiniowaniem, w obrębie funkcji main występuje deklaracja zapowiadająca. Deklaracja ta ma postać nagłówka definicji, z którego usunięto listę nazw parametrów i ich deklaracje.

W przypadku gdy wykonywanie funkcji nie kończy się wykonaniem instrukcji return zawierającej wyrażenie, określenie typu rezultatu jest zbyteczne. W takim przypadku zbędne jest także użycie deklaracji zapowiadającej. W języku **C** dopuszczono także pominięcie określenia typu rezultatu funkcji jeśli jest nim (int) oraz dopuszczono pominięcie deklaracji tych parametrów funkcji, które są typu (int). Wobec tych uproszczeń następujący program którego wykonanie powoduje wyprowadzenie liczby 13

```
int _13 = -13;
int
main()
{
    int Negate();
    printf("%d", Negate(_13));
}
```

```
int
Negate(Par)
    int Par;
{
    return -Par;
}
```

może zostać przedstawiony w postaci uproszczonej

```
int _13 = -13;

main()
{
    printf("%d", Negate(_13));
}

Negate(Par)
{
    return -Par;
}
```

Sposób wykonywania programu nie zależy od kolejności w jakiej występują w nim definicje funkcji. Tym niemniej wymaga się, aby odwołanie do funkcji udostępniających rezultaty innego typu jak (int), były poprzedzone deklaracjami zapowiadającymi tych funkcji.

Deklaracja zapowiadająca ma taką postać jak nagłówek funkcji, z którego usunięto listę i deklaracje parametrów, ale który dodatkowo poprzedzono słowem extern. Jeśli taka deklaracja znajduje się wewnątrz definicji funkcji, to wspomniane słowo może być pominięte. W następującym programie występują m.in. definicje funkcji First i Second oraz deklaracje zapowiadające tych funkcji. Miejsca wystąpienia deklaracji zapowiadających oznaczono komentarzami.

```
char Name[2] = "bj";
extern char * First ( );          /* First */

main( )
{
    printf("%c", *First( ));
}

char *
First( )
{
    char *Second( );              /* Second */
```

```
    printf("%c", *Second(Name));
    return Name;
}

char *
Second (Initials)
    char Initials[2];
{
    return Initials + 1;
}
```

Wykonanie programu powoduje wyprowadzenie napisu jb.

Deklarowanie typów

Omówienie zasad deklarowania *obiektów programowych* nie byłoby kompletne, gdyby przemilczano możliwość deklarowania *typów*. Deklaracja typu ma w języku **C** postać zbliżoną do deklaracji zmiennej. Przekształcenie deklaracji zmiennej w deklarację typu polega na poprzedzeniu oznaczenia typu słowem typedef i potraktowaniu identyfikatorów zmiennych jako nazw właśnie deklarowanych typów. W tym sensie deklaracja

```
typedef char *PTR,
             VEC[3];
```

definiuje typ (PTR) identyfikujący wskazania danych typu (char) oraz typ (VEC) identyfikujący 3-elementowe tablice danych typu (char). Po zadeklarowaniu omawianych typów można posługiwać się takimi deklaracjami jak np.

```
PTR Arr[2], ChrPtr;
VEC *Ref,
```

Pierwsza z nich deklaruje 2-elementową tablicę zmiennych typu (PTR), a więc tablicę wskazań danych typu (char) oraz zmienną ChrPtr wskazującą takie dane, natomiast druga deklaruje wskazanie danych typu (VEC), a więc wskazanie 3-elementowych tablic typu (char). Nietrudno stąd wywnioskować, że przytoczone deklaracje zmiennych Arr, ChrPtr i Ref są w istocie równoważne deklaracjom

```
char *Arr[2], ChrPtr;
char (*Ref)[3];
```

Jak można się było spodziewać, deklarowanie typów nie musi ograniczać się do typów wskazujących i tablicowych. Przykładem poprawnej deklaracji typu strukturowego może być bowiem deklaracja

```
typedef struct List{
        int Data;
        struct List *Link;
      } LIST;
```

w której LIST jest nazwą typu (struct List). W zasięgu takiej deklaracji typu przykładowa deklaracja

```
LIST Str[3], *Ref;
```

jest traktowana dokładnie tak jak deklaracja

```
struct List Str[3], *Ref;
```

4

KONSTRUOWANIE I WYZNACZANIE WARTOŚCI WYRAŻEŃ

Podstawowymi elementami wyrażeń języka **C** są *wyrażenia pierwotne*. Jak wynika z zestawienia przytoczonego w tabl. 4.1 wyrażeniem pierwotnym jest

- identyfikator, np. Volume, Fun,
- literał, np. 13, "jb", ' \n',
- dowolne wyrażenie ujęte w nawiasy okrągłe, np. (a+b),
- wyrażenie reprezentujące funkcję, po którym następuje lista argumentów ujęta w nawiasy okrągłe, np. Fun(a,b),
- wyrażenie pierwotne, po którym następuje wyrażenie ujęte w nawiasy kwadratowe, np. Arr[i+2],
- l-wyrażenie pierwotne reprezentujące strukturę, bezpośrednio po którym następuje znak . (kropka) i identyfikator pola struktury, np. Str.Chr,
- wyrażenie pierwotne reprezentujące wskazanie struktury, po którym bezpośrednio następuje symbol −> (*minus, większe*) i identyfikator pola wskazywanej struktury, np. Ptr−>Chr.

W przedstawionym zestawieniu Arr[i] reprezentuje i-ty element tablicy, Fun(a,b) reprezentuje rezultat funkcji, Arr[i+2] reprezentuje element tablicy, a Str. Chr i Ptr−>Chr reprezentuje pole struktury. Użyte w jednej z definicji pojęcie *l-wyrażenie* dotyczy takiego wyrażenia, które reprezentuje zmienną prostą i należy do kategorii wyrażeń zdefiniowanych w tabl. 4.2. Takie wyrażenie może wystąpić z lewej strony znaku albo symbolu przypisania i właśnie temu (l od lewa) zawdzięcza swoją nazwę.

Zastosowanie się do schematów przedstawionych w tabl. 4.1 i 4.2 nie wystarcza do konstruowania wyrażeń poprawnych. W celu uniknięcia błędów programowania należy mieć na uwadze następujące reguły dodatkowe

- nie jest l-wyrażeniem napis, który reprezentuje wskazanie obiektu, np. &Fix,
- w napisie o postaci

  wyrażenie-pierwotne (*lista-wyrażeń*)

wyrażenie-pierwotne powinno reprezentować funkcję, np. Fun(a,b),
– w napisie o postaci

wyrażenie-pierwotne [*wyrażenie*]

dokładnie jedno z wymienionych *wyrażeń* powinno być wyrażeniem wskazującym, reprezentującym wskazanie elementu tablicy, np. 2["jan"],
– w napisie o postaci

l-wyrażenie-pierwotne . *identyfikator*

l-wyrażenie-pierwotne powinno reprezentować strukturę albo unię, a *identyfikator* powinien być nazwą pola tej struktury albo unii,
– w napisie o postaci

l-wyrażenie –> *identyfikator*

l-wyrażenie powinno reprezentować wskazanie struktury albo unii, a *identyfikator* powinien być nazwą pola tej struktury albo unii,
– w napisie o postaci

* *wyrażenie*

wyrażenie powinno być wyrażeniem wskazującym zmienną prostą, np. zmienną stanowiącą element jednowymiarowej tablicy albo takie pole struktury albo unii, które nie jest tablicą, strukturą ani unią.

Po zdefiniowaniu pojęć *wyrażenie-pierwotne* i *l-wyrażenie* można przystąpić do zdefiniowania pojęcia *wyrażenie* w jego najogólniejszej postaci. Dokonano tego za pomocą tablicy 4.3, w której zdefiniowano ponadto pojęcia operacji i przypisania.

Jak wynika z przytoczonego zestawienia, każde wyrażenie pierwotne także jest wyrażeniem. Oznacza to, że wyrażeniami są m.in. napisy: Vol, "jb", 'j', 44, (a+2), Fun(x,y), Arr[3] [4], Str.Fix i Ref–>Fix.

Wyrażeniem jest dowolne wyrażenie wskazujące poprzedzone operatorem wyłuskania * (*gwiazdka*). Jeśli np. wyrażenie Exp reprezentuje wskazanie pewnego obiektu, to wyrażenie *Exp reprezentuje ten obiekt.

Wyrażeniem jest l-wyrażenie poprzedzone operatorem wskazania & (*ampersand*). Jeśli np. Nam reprezentuje pewien obiekt, to &Nam reprezentuje wskazanie tego obiektu.

Wyrażeniem jest wyrażenie poprzedzone operatorem zmiany znaku – (*minus*). Jeśli np. Exp jest wyrażeniem, to –Exp reprezentuje daną o wartości 0–Exp.

Wyrażeniem jest wyrażenie poprzedzone operatorem zaprzeczenia ! (*wykrzyknik*). Jeśli np. Exp jest wyrażeniem, to !Exp reprezentuje daną o wartości 1 – jeśli Exp reprezentowało daną o wartości 0, albo reprezentuje daną o wartości 0 – w przeciwnym razie.

Wyrażeniem jest wyrażenie poprzedzone operatorem negacji ~(*tylda*). Jeśli np. Exp reprezentuje daną 16-bitową, to ~Exp reprezentuje daną powstałą z zanegowania wszystkich bitów danej Exp.

Wyrażeniem jest l-wyrażenie poprzedzone operatorem preinkrementacji ++ (*plus, plus*). Jeśli np. Exp jest l-wyrażeniem, to ++Exp reprezentuje daną o wartości Exp+1. Skutkiem ubocznym wykonania omawianej operacji jest zwiększenie o 1 zmiennej reprezentowanej przez Exp.

Tablica 4.1 Definicja pojęcia wyrażenie-pierwotne

```
wyrażenie-pierwotne:
  identyfikator
  literał
  ( wyrażenie )
  wyrażenie-pierwotne ( lista-wyrażeń )
  wyrażenie-pierwotne [ wyrażenie ]
  l-wyrażenie-pierwotne . identyfikator
  wyrażenie-pierwotne -> identyfikator
```

Tablica 4.2 Definicja pojęcia l-wyrażenie

```
l-wyrażenie:
  identyfikator
  wyrażenie-pierwotne [ wyrażenie ]
  l-wyrażenie-pierwotne . identyfikator
  wyrażenie-pierwotne -> identyfikator
  * wyrażenie
  ( l-wyrażenie )
```

Tablica 4.3 Definicja pojęcia wyrażenie

```
wyrażenie:
  wyrażenie-pierwotne
  * wyrażenie
  & l-wyrażenie
  - wyrażenie
  ! wyrażenie
  ~ wyrażenie
  ++ l-wyrażenie
  -- l-wyrażenie
  l-wyrażenie ++
  l-wyrażenie --
  sizeof ( nazwa-typu )
  sizeof wyrażenie
  ( nazwa-typu ) wyrażenie
  wyrażenie operator-dwuargumentowy wyrażenie
  wyrażenie ? wyrażenie : wyrażenie
  l-wyrażenie operator-przypisania wyrażenie
  wyrażenie , wyrażenie

operator-dwuargumentowy:
  *  /  %
  +  -
  <<   >>
  <  >  <=  >=
  ==   !=
  &
  ^
  |
  &&
  ||

operator-przypisania:
  =  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=
```

Wyrażeniem jest l-wyrażenie poprzedzone operatorem predekrementacji −− (*minus, minus*). Jeśli np. Exp jest l-wyrażeniem, to −−Exp reprezentuje daną o wartości Exp−1. Skutkiem ubocznym wykonania omawianej operacji jest zmniejszenie o 1 zmiennej reprezentowanej przez Exp.

Wyrażeniem jest l-wyrażenie bezpośrednio po którym występuje operator postinkrementacji ++ (*plus, plus*). Jeśli np. Exp jest l-wyrażeniem, to Exp++ reprezentuje daną o wartości Exp. Skutkiem ubocznym wykonania omawianej operacji jest zwiększenie o 1 zmiennej reprezentowanej przez Exp.

Wyrażeniem jest l-wyrażenie bezpośrednio po którym występuje operator postdekrementacji −− (*minus, minus*). Jeśli np. Exp jest l-wyrażeniem, to Exp−− reprezentuje daną o wartości Exp. Skutkiem ubocznym omawianej operacji jest zmniejszenie o 1 zmiennej reprezentowanej przez Exp.

Wyrażeniem jest wyrażenie poprzedzone operatorem konwersji *(nazwa-typu)*. Jeśli Exp jest wyrażeniem reprezentującym pewną daną, to *(nazwa-typu)*Exp jest wyrażeniem reprezentującym daną typu *(nazwa-typu)*. Nazwą typu jest napis składający się z oznaczenia typu i deklaratora. Jeśli np. Exp jest wyrażeniem typu (int *), to wyrażenie (char *)Exp reprezentuje daną typu (char *).

Uwaga: W systemie HiSoft napis *(nazwa-typu)* występujący w znaczeniu operatora konwersji musi być poprzedzony słowem kluczowym cast, np. cast(char *)Exp.

Wyrażeniem jest nazwa typu ujęta w nawiasy okrągłe i poprzedzona operatorem rozmiaru sizeof. Jeśli np. *Nam* jest nazwą typu, to sizeof(*Nam*) jest wyrażeniem reprezentującym daną o wartości równej liczbie bajtów niezbędnych do przechowania w pamięci wewnętrznej danej typu (*Nam*).

Uwaga: W systemie HiSoft nazwa typu musi być słowem kluczowym oznaczającym typ, np. int, albo identyfikatorem typu.

Wyrażeniem jest wyrażenie poprzedzone operatorem rozmiaru sizeof. Jeśli np. Exp jest wyrażeniem reprezentującym pewną daną, to sizeof Exp jest wyrażeniem reprezentującym daną o wartości równej liczbie bajtów niezbędnych do przechowania w pamięci wewnętrznej danej takiego typu jak Exp.

Uwaga: W systemie HiSoft operator sizeof nie może dotyczyć wyrażeń.

Wyrażeniem jest napis o postaci

wyrażenie operator-dwuargumentowy wyrażenie

W napisie tym określenie *operator-dwuargumentowy* dotyczy jednego z następujących operatorów: * (*mnożenia*), | (*dzielenia*), % (*reszty z dzielenia*), + (*dodawania*), − (*odejmowania*), <<(*przesunięcia w lewo*), >> *przesunięcia w prawo*), < (*mniejszości*), > (*większości*), <= (*mniejszości lub równości*), >= (*większości lub równości*), == (*równości*), != (*nierówno-*

ści, & (*iloczynu logicznego bitów*), ^(różnicy symetrycznej bitów), | (*sumy logicznej bitów*), && (*iloczynu logicznego*) i || (*sumy logicznej*).

a * b Rezultatem operacji jest iloczyn argumentów. Mnożenie może dotyczyć jedynie danych arytmetycznych.

a / b Rezultatem operacji jest iloraz argumentów. Dzielenie może dotyczyć tylko danych arytmetycznych.

a % b Rezultatem operacji jest reszta z dzielenia pierwszego argumentu przez drugi. Jako definicję operacji można przyjąć formułę a % b = a − (a / b) * b.

a + b Rezultatem operacji jest suma argumentów. Jeśli jeden z argumentów jest typu wskazującego i reprezentuje wskazanie pewnego elementu Arr[i] tablicy Arr, a drugi jest typu całkowitego i reprezentuje daną o wartości j, to suma reprezentuje wskazanie elementu Arr[i+j].

a − b Rezultatem operacji jest różnica argumentów. Jeśli pierwszy argument jest typu wskazującego i reprezentuje wskazanie pewnego elementu Arr[i] tablicy Arr, a drugi jest typu całkowitego i reprezentuje daną o wartości j, to różnica reprezentuje wskazanie elementu Arr[i−j]. Jeśli natomiast oba argumenty są typu wskazującego i reprezentują wskazania elementów Arr[i] i Arr[j], to ich różnica reprezentuje daną o wartości i−j.

a<< b Rezultatem operacji jest dana powstała z przesunięcia reprezentacji pierwszego argumentu w lewo o liczbę pozycji określoną przez drugi argument.

a >> b Rezultatem operacji jest dana powstała z przesunięcia reprezentacji pierwszego argumentu w prawo o liczbę pozycji określoną przez drugi argument.

a < b Rezultatem operacji jest dana o wartości 0 (fałsz) albo 1 (prawda), wyrażająca prawdziwość zdania "pierwszy argument jest mniejszy niż drugi".

a > b Rezultatem operacji jest dana o wartości 0 (fałsz) albo 1 (prawda), wyrażająca prawdziwość zdania "pierwszy argument jest większy niż drugi".

a <= b Rezultatem operacji jest dana o wartości 0 (fałsz) albo 1 (prawda), wyrażająca prawdziwość zdania "pierwszy argument jest mniejszy niż drugi albo mu równy."

a >= b Rezultatem operacji jest dana o wartości 0 (fałsz) albo 1 (prawda), wyrażająca prawdziwość zdania "pierwszy argument jest większy niż drugi albo mu równy".

a == b Rezultatem operacji jest dana o wartości 0 (fałsz) albo 1 (prawda), wyrażająca prawdziwość zdania "pierwszy argument jest równy drugiemu".

a != b Rezultatem operacji jest dana o wartości 0 (fałsz) albo 1 (prawda), wyrażająca prawdziwość zdania "pierwszy argument nie jest równy drugiemu".

a & b Rezultatem operacji jest iloczyn logiczny bitów, wyznaczony równolegle na odpowiadających sobie parach bitów argumentów.

a ^ b Rezultatem operacji jest różnica symetryczna bitów (suma modulo 2), wyznaczona równolegle na odpowiadających sobie parach bitów argumentów.

a | b Rezultatem operacji jest suma logiczna bitów, wyznaczona równolegle na odpowiadających sobie parach bitów argumentów.

a && b Rezultatem operacji jest dana o wartości 0 (fałsz) albo 1 (prawda) wyrażająca prawdziwość zdania "każdy z argumentów ma wartość różną od 0". Należy zaznaczyć, że operacja jest wykonywana od lewej do prawej i jeśli jest a= =0, to argument b nie jest rozpatrywany.

a || b Rezultatem operacji jest dana o wartości 0 (fałsz) albo 1 (prawda) wyrażająca prawdziwość zdania "przynajmniej jeden z argumentów ma wartość różną niż 0". Należy zaznaczyć, że operacja jest wykonywana od lewej do prawej i jeśli a! =0, to argument b nie jest rozpatrywany.

Wyrażeniem jest napis o postaci

l-wyrażenie operator-przypisania wyrażenie

W napisie tym określenie *operator-przypisania* dotyczy znaku przypisania = (*równa się*) albo jednego z symboli przypisania: *=, |=, %=, +=, -=, <<=, >>=, &=, ^= i |=. Wykonanie operacji a = b powoduje przypisanie zmiennej reprezentowanej przez a, danej reprezentowanej przez b. Rezultatem operacji jest dana przypisana zmiennej a. Wykonanie operacji a ⊙ = b, w której napis ⊙ reprezentuje jeden z dopuszczalnych znaków symbolu przypisania, jest równoważne wykonaniu operacji a = a ⊙ b. Wykonanie tej operacji powinno zostać zrealizowane w taki sposób, aby wyznaczenie wartości wyrażenia a było jednokrotne.

Wyrażeniem jest napis o postaci

wyrażenie ? wyrażenie-1 : wyrażenie-2

Jeśli *wyrażenie* ma wartość różną niż 0, to rezultatem trójargumentowej operacji ?: jest dana reprezentowana przez *wyrażenie-1*, a w przeciwnym razie jest dana reprezentowana przez *wyrażenie-2*. Zaleca się, aby *wyrażenie-1* było takiego samego typu jak *wyrażenie-2*.

Wyrażeniem jest napis

wyrażenie-1 , wyrażenie-2

Wykonanie operacji połączenia , (*przecinek*) powoduje wyznaczenie wartości *wyrażenia-1*, a następnie wyznaczenie wartości *wyrażenia-2*. Rezultatem operacji połączenia jest dana reprezentowana przez *wyrażenie-2*.

Uwaga: W systemie HiSoft nie implementowano operatora połączenia.

Po skonstruowaniu wyrażenia poprawnego istotne staje się określenie sposobu jego interpretowania. Zależy to od ustalonych w języku *priorytetów* i *wiązań operatorów*, przytoczonych w tabl. 4.4. Operacje związane z operatorami o wyższym priorytecie są wykonywane w pierwszej kolejności. Jeśli pewnego argumentu operacji dotyczą dwa operatory o równym priorytecie, to kolejność wykonywania operacji jest określona przez wiązanie. Uwzględnienie priorytetu i wiązań operatorów umożliwia w wielu wypadkach wyeliminowanie z wyrażeń zbędnych nawiasów okrągłych.

W szczególności poprawne jest np. wyrażenie

```
a -= b = -c++ + d - e
```

całkowicie pozbawione nawiasów okrągłych. W wyrażeniu tym argumentu c dotyczą dwa operatory o równym priorytecie: operator zmiany znaku (-) i operator postinkrementacji (++). Ponieważ wiązanie każdego z tych operatorów jest prawostronne, w pierwszej kolejności zostanie wykonana operacja postinkrementacji, a dopiero potem zmiana znaku. Z tego powodu rozpatrywane wyrażenie jest traktowane tak jakby miało postać

```
a -= b = ( -( c ++)) + d - e
```

Ponieważ priorytety operatorów dodawania (+) i odejmowania (-) są wyższe niż priorytet operatora przypisania (=), można wyrazić to posługując się parą dodatkowych nawiasów

```
a -= b = ((-(c++)) + d - e)
```

Operacje dodawania (+) i odejmowania (-) występujące w obrębie zewnętrznych nawiasów mają równy priorytet. Ponieważ operatory dodawania (+) i odejmowania (-) wiążą argumenty lewostronnie, rozpatrywane wyrażenie jest interpretowane tak jak wyrażenie

```
a -= b - (((-(c++)) + d) - e)
```

Dla odmiany operatory przypisania wiążą swoje argumenty prawostronnie, a więc ostatecznie można wywnioskować, że pierwotne wyrażenie

```
a -= b =-c++ + d - e
```

Tablica 4.4 Priorytety i wiązania operatorów

Priorytet	Wiązanie	Operator
15	lewe	() [] -> .
14	prawe	! ~ ++ -- - (typ) * & sizeof
13	lewe	* / %
12	lewe	+ -
11	lewe	<< >>
10	lewe	< <= > >=
9	lewe	== !=
8	lewe	&
7	lewe	^
6	lewe	|
5	lewe	&&
4	lewe	||
3	prawe	?:
2	prawe	= *= /= %= += -= <<= >>= &= ^= |=
1	lewe	,

jest interpretowane tak jak wyrażenie

```
(a -= (b = (((-(c++)) + d) - e)))
```

Jeśli przyjąć, że rozpatrywane wyrażenie zostanie użyte w programie

```
int a = 10,
    b = 20,
    c = 30,
    d = 40,
    e = 50;

main()
{
    a -= b = -c++ + d - e;
    printf("a=%d b=%d c=%d", a,b,c);
}
```

to jego wykonanie spowoduje wyprowadzenie napisu

```
a=50 b=-40 c= 31
```

Należy wyjaśnić, że posłużenie się priorytetami i wiązaniami operatorów umożliwia jedynie wyodrębnienie z wyrażenia poszczególnych operacji. Nie wyklucza to jednak faktu, iż operacje, które są łączne i przemienne, takie jak np. dodawanie, mogą zostać wykonane w innej kolejności niż wynika to z rozmieszczenia nawiasów. W szczególności wartość wyrażenia

```
a + b + c
```

zinterpretowanego jako

```
(a + b) + c
```

może zostać wyznaczona w taki sposób jak wartość wyrażenia

```
a + (b+ c)
```

Oznacza to, iż w języku **C** w przypadku działań łącznych i przemiennych nie jest respektowane rozmieszczenie nawiasów. Na szczęście nie ma to w praktyce istotnego znaczenia i ujawnia się tylko w programach z "udziwnionymi" skutkami ubocznymi.

Uwaga: W systemie HiSoft obowiązuje pełne respektowanie nawiasów.

Jeśli skonstruowane wyrażenie jest poprawne pod względem składniowym i właściwie zinterpretowane na podstawie informacji o wiązaniach i priorytetach użytych w nim operatorów, to wyznaczenie jego wartości przebiega w sposób znany z innych języków programowania. Trudność mogą sprawiać jedynie *jawne* i *niejawne konwersje danych.*

Jawne konwersje danych są wyrażone za pomocą operatora konwersji, który w systemie HiSoft musi być poprzedzony słowem kluczowym cast.

W następującym programie

```
char Arr[2] [2] = { 'J','a','n','B' },
     *Ptr;

main()
{
   typedef char *ChrPtr;

   Ptr = cast(ChrPtr) (Arr + 1) + 1;
   printf("%c%c", **Arr,*Ptr);
}
```

zastosowano operator konwersji do wyrażenia (Arr + 1). Powoduje to, że wspomniane wyrażenie, które reprezentuje daną typu (char (*) [2]) zostaje przekształcone w wyrażenie reprezentujące daną (char *), a więc w wyrażenie takiego samego typu jakiego jest zmienna Ptr. Dzięki temu poprawne jest wykonanie przypisania. Należy nadmienić, że gdyby w przytoczonym programie, którego wykonanie powoduje wyprowadzenie napisu JB, dokonano zmiany operatora cast(ChrPtr) na operator wyłuskania * (*gwiazdka*), to rezultat programu byłby taki sam. Świadczy to o istotnej roli użytej tu konwersji.

Wymaganie, aby przypisywana dana była takiego samego typu jak zmienna, której dotyczy przypisanie jest egzekwowane w odniesieniu do obiektów wskazujących, ale nie jest egzekwowane w odniesieniu do obiektów całkowitych. W szczególności, jeśli dana typu (char) zostanie przypisana zmiennej typu (int) albo (unsigned), to będzie to polegać na uzupełnieniu jej bardziej znaczącym bajtem składającym się z samych zer. Natomiast gdy dana typu (int) albo (unsigned) zostanie przypisana zmiennej typu (char), to sprowadzi się to do przypisania dolnego bajtu danej. Natomiast przypisanie danych typu (int) zmiennym typu (unsigned) lub odwrotnie, dokonują się w taki sposób, że być może nastąpi zmiana wartości, ale nie nastąpi zmiana reprezentacji danej.

Niejawne konwersje mogą występować nie tylko podczas wykonywania przypisań. W trakcie wykonywania operacji arytmetycznych obowiązują bowiem następujące ustalenia.

- Jeśli pewien argument jest typu (*char*), to jest poddawany konwersji na daną typu (*int*).
- Jeśli tylko jeden z argumentów operacji dwuargumentowej jest typu (*unsigned*), to drugi jest poddawany konwersji na daną typu (*unsigned*).
- W pozostałych przypadkach oba argumenty muszą być typu (*int*) i takiego właśnie typu jest rezultat operacji.

Uwaga: Przytoczone zasady konwersji danych całkowitych ograniczono do tych, które obowiązują w systemie HiSoft.

5
WYKONYWANIE INSTRUKCJI

Instrukcjami języka **C** są instrukcje wyrażeniowe, instrukcje puste i grupujące oraz instrukcje rozpoczynające się od słowa kluczowego. Każda instrukcja programu, z wyjątkiem instrukcji grupującej stanowiącej ciało funkcji, może być poprzedzona etykietą albo grupą etykiet. Znakiem oddzielającym etykietę od instrukcji albo od innej etykiety jest : (*dwukropek*). Przykładem instrukcji poprzedzonej etykietą jest

```
fin : return a + b;
```

Instrukcja wyrażeniowa

Instrukcja ta ma postać

exp ;

w której exp jest dowolnym wyrażeniem. Wykonanie instrukcji wyrażeniowej powoduje wyznaczenie danej reprezentowanej przez wyrażenie *exp*, a następnie zignorowanie jej. W tym sensie wykonanie instrukcji

```
2 + (a = 5);
```

powoduje wyznaczenie danej o wartości 7 i zignorowanie jej. Podczas wyznaczania tej danej następuje przypisanie zmiennej a danej o wartości 5. Oczywiście skutek wykonania rozpatrywanej instrukcji jest taki sam jak skutek wykonania instrukcji

```
a = 5;
```

Instrukcja pusta

Instrukcja ta ma postać

;

Wykonanie instrukcji pustej nie wywołuje żadnych skutków.

```
main()
{
   ;
}
```

Ciało funkcji main zawiera instrukcję pustą.

Instrukcja grupująca

Instrukcja ta ma postać

```
{
   zestaw-deklaracji
   zestaw-instrukcji
}
```

Wykonanie instrukcji grupującej składa się z wykonania *zestawu-instrukcji*. Deklaracje zawarte w *zestawie-deklaracji* ograniczają zasięg deklaracji występujących poza instrukcją grupującą.
Uwaga: W systemie HiSoft *zestaw-deklaracji* może występować jedynie w takiej instrukcji grupującej, która stanowi ciało funkcji.

```
int Fix = 3;
char Chr = 'J';

main()
{
   char Fix;

   Fix = 0x42;
   {
   printf("%c%c", Chr,Fix);
   }
}
```

W obrębie instrukcji grupującej stanowiącej ciało funkcji main, deklaracja zmiennej Fix typu (char) przesłania deklarację zmiennej Fix typu (int). Wykonanie programu powoduje wyprowadzenie napisu JB.

Deklaracja zmiennej Chr obowiązuje w obrębie każdej z 2 użytych w programie instrukcji grupujących.

Instrukcja przejścia

Instrukcja ta ma postać

goto *lab* ;

w której napis *lab* jest identyczny z etykietą opatrującą pewną instrukcję tej samej funkcji, w której wystąpiła rozpatrywana instrukcja goto.

```
main()
{
   {
      goto Fin;
   }
   Fin: ;
}
```

Wykonanie programu instrukcji goto powoduje przejście do wykonywania instrukcji pustej opatrzonej etykietą Fin.

Instrukcja warunkowa

Instrukcja ta ma postać

```
if ( wyrażenie )
  instrukcja_p
else
  instrukcja_f
```

albo postać

```
if ( wyrażenie )
  instrukcja_p
```

równoważną instrukcji

```
if ( wyrażenie )
  instrukcja_p
else
  ;
```

Wykonanie instrukcji warunkowej w pierwszej z wymienionych postaci składa się z wyznaczenia wartości danej reprezentowanej przez *wyrażenie*, a następnie z wykonania $instrukcji_p$ – jeśli wartość ta jest różna od 0, albo wykonania $instrukcji_f$ – w przeciwnym razie. Przyjmuje się, że jeśli $instrukcja_p$ lub $instrukcja_f$ jest instrukcją warunkową, to słowo kluczowe else jest kojarzone z najbliższym poprzedzającym ją, jeszcze nie skojarzonym, słowem if.

```
int  Num = 2,
     Dec;

main()
{
   if(Dec = Num)
     if(Dec == Num)
       Num = 4;
     else {
       Num *= Dec;
       printf("%d %d", Num,Dec);
     }
   else
```

```
        printf("%d", Num);
      printf("ok");
}
```

Wykonanie programu powoduje wyprowadzenie napisu ok. Należy odnotować, że w wyrażeniu Dec = Num występuje operator przypisania, a w wyrażeniu Dec == Num występuje operator relacji. Pierwsze z wymienionych wyrażeń reprezentuje daną o wartości 2, a drugie reprezentuje daną o wartości 1.

Instrukcja decyzyjna

Instrukcja ta ma postać

switch (*wyrażenie*)
  instrukcja

w której *instrukcja* reprezentuje dowolną instrukcję nazywaną dalej *obiektem decyzji.* W obiekcie decyzji (który jest zazwyczaj instrukcją grupującą) mogą wystąpić przedrostki instrukcji o postaci

case *wyrażenie-stałe* :

oraz może wystąpić co najwyżej jeden przedrostek o postaci

default :

także poprzedzający pewną instrukcję. Wymaga się, by *wyrażenie* oraz każde z *wyrażeń-stałych* było typu całkowitego.

Wykonanie instrukcji decyzyjnej rozpoczyna się od wyznaczenia wartości *wyrażenia* występującego po słowie kluczowym switch. Wartość tego wyrażenia jest następnie porównywana z wartościami *wyrażeń-stałych* występujących w przedrostkach case. Jeśli dla pewnego przedrostka zostanie stwierdzona wartość, to wykonywanie programu będzie kontynuowane od instrukcji poprzedzonej tym przedrostkiem. Jeśli równość nie zostanie stwierdzona dla żadnego przedrostka, to wykonywanie programu będzie kontynuowane począwszy od instrukcji poprzedzonej przedrostkiem default:. Jeśli obiekt decyzji nie zawiera takiego przedrostka, to wykonywanie instrukcji decyzyjnej zostanie uznane za zakończone.

```
main()
{
   Star(1); Star(0);
}

Star(Count)
   int Count;
{
   switch(Count){
      case 1 :
```

```
            printf("J");
        case 2 :
            printf("an");
            break;
        default :
            printf("B");
    }
}
```

Wykonanie programu powoduje wyprowadzenie napisu JanB.

Instrukcje powtarzające

W języku **C** istnieją trzy odmiany instrukcji powtarzających: instrukcja *while*, instrukcja *do* i instrukcja *for*. Opis tych instrukcji zostanie przedstawiony za pomocą instrukcji warunkowych i instrukcji przejścia.

Instrukcja while

Instrukcja ta ma postać

```
while ( wyrażenie )
    instrukcja
```

i jest wykonywana tak jak podprogram otwarty

```
Lab:  if(wyrażenie){
          instrukcja
          goto Lab;
      }
```

Inaczej to ujmując, instrukcja jest wykonywana tak długo, jak długo *wyrażenie* ma wartość różną od 0. Określenie wartości wyrażenia odbywa się przed każdym wykonaniem instrukcji.

```
char *Ptr = "Jan Bielecki";

main()
{
    while(*Ptr)
        printf("%c", *Ptr++);
}
```

Wykonanie programu powoduje wyprowadzenie napisu Jan Bielecki.

Instrukcja do

Instrukcja ta ma postać

```
do
    instrukcja
while ( wyrażenie ) ;
```

i jest wykonywana tak jak podprogram otwarty

```
Lab: instrukcja
     if(wyrażenie)
       goto Lab;
```

Inaczej to ujmując, instrukcja jest wykonywana tak długo, jak długo *wyrażenie* ma wartość różną od 0. Określenie wartości wyrażenia odbywa się po każdym wykonaniu instrukcji.

```
char *Ptr = "Jan Bielecki";

main()
{
   do
      printf("%c", *Ptr);
   while(*++Ptr);
}
```

Wykonanie programu powoduje wyprowadzenie napisu Jan Bielecki.

Instrukcja for

Instrukcja ta ma postać

for ($wyrażenie_a$; *wyrażenie* ; $wyrażenie_b$)
  instrukcja

i jest wykonywana tak jak podprogram otwarty

```
wyrażenie_a;
Lab: if(wyrażenie){
       instrukcja
       wyrażenie_b;
       goto Lab;
     }
```

Inaczej to ujmując, bezpośrednio po wykonaniu instrukcji wyrażeniowej utworzonej z $wyrażenia_a$ tak długo następuje wykonywanie instrukcji utworzonej z $wyrażenia_b$, jak długo *wyrażenie* ma wartość różną od 0.

Każde z wymienionych *wyrażeń* może być puste. Jeśli dotyczy to środkowego, to traktuje się je tak, jakby było liczbą 1.

```
char *Ptr;

main()
{
   for(Ptr = "Jan Bielecki"; *Ptr; )
      printf("%c", *Ptr++);
}
```

Wykonanie programu powoduje wyprowadzenie napisu Jan Bielecki.

Instrukcja kontynuowania

Instrukcja ta ma postać

```
continue ;
```

i może wystąpić w obrębie dowolnej instrukcji powtarzającej.

Wykonanie instrukcji kontynuowania powoduje kontynuowanie wykonywania najwęższej obejmującej ją instrukcji powtarzającej. Kontynuacja rozpoczyna się od miejsca, w którym są badane kryteria powtarzania. Oznacza to, że w każdym z następujących przypadków, wykonanie instrukcji continue występujące w miejscu oznaczonym komentarzem mogłoby zostać zastąpione wykonaniem instrukcji

```
goto Next;
```

Instrukcja while

```
while( ... ){
   /* ... */
Next: ;
}
```

Instrukcja do

```
do {
   /* ... */
Next: ;
}
```

Instrukcja for

```
for( ... ){
   /* ... */
Next: ;
}
```

Użycie instrukcji kontynuowania powinno być w programach zredukowane do minimum, ponieważ w większości przypadków czyni to programy prostszymi. W szczególności funkcja Spaces następującego programu zliczającego znaki różne od spacji zawarte w ciągu "Jan & Ewa"

```
main()
{
   printf("%d", Spaces("Jan & Ewa"));
}
```

```
int
Spaces(Ptr)
   char *Ptr;
{
   int Count;

   Count = 0;
   while(*Ptr){
      if(*Ptr++ == ' ')
        continue;
      Count++;
   }
   return Count;
}
```

mogłaby zostać zapisana prościej jako

```
int
Spaces(Ptr)
   char *Ptr;
{
   int Count;
   Count = 0;
   while(*Ptr)
      Count += *Ptr++ != ' ';
   return Count;
}
```

Instrukcja zaniechania

Instrukcja ta ma postać

```
break ;
```

i może wystąpić w obrębie dowolnej instrukcji powtarzającej albo decyzyjnej.

Wykonanie instrukcji zaniechania powoduje zaniechanie dalszego wykonywania najwęższej z obejmujących ją wymienionych instrukcji i przejście do wykonywania instrukcji po niej następnej.

```
char *Ptr = "Jan Bielecki";

main()
{
   for(;;){
      printf("%c", *Ptr);
      if(!*++Ptr)
        break;
   }
}
```

Wykonanie programu powoduje wyprowadzenie napisu Jan Bielecki.

Instrukcja powrotu

Instrukcja ta ma postać

return *exp* ;

albo postać

return ;

Wykonanie instrukcji powrotu powoduje zakończenie wykonywania zawierającej ją funkcji. Jeśli instrukcja powrotu zawiera wyrażenie *exp*, to dana reprezentowana przez to wyrażenie stanowi udostępniany rezultat funkcji. Wymaga się, aby daną tą nie była tablica, struktura, unia ani funkcja.

```
char *Ptr = "Jan Bielecki";

main()
{
   while(1){
      printf("%c", *Ptr++);
      if(!Ptr[0])
        return;
   }
}
```

Wykonanie programu powoduje wyprowadzenie napisu Jan Bielecki.

6
WYWOŁYWANIE FUNKCJI

Wywołanie funkcji ma postać

fun(arg_1,arg_2, ... , arg_n)

W napisie tym fun jest wyrażeniem reprezentującym funkcję, najczęściej jej identyfikatorem, natomiast arg_i (i=1,2, ... ,n) są argumentami wywołania.

```
main()
{
   int print();

   output(13,print);
}

output(Num,Fun)
   int Num,(*Fun)();
{
   (*Fun)("%d", Num);
}

print(Fmt,Val)
   char *Fmt;
   int Val;
{
   printf(Fmt,-Val);
}
```

Wykonanie programu powoduje wyprowadzenie liczby −13. Argument print zostaje niejawnie przekształcony na wskazanie funkcji print, dlatego *Fun reprezentuje funkcję printf i instrukcja

```
(*Fun)("%d", Num);
```

jest wykonywana tak jak instrukcja

```
printf("%d", Num);
```

W chwili wywołania funkcji następuje skojarzenie jej parametrów z argumentami wywołania. Skojarzenie to dokonuje się przez wartość i polega na tym, że parametry są traktowane tak jak zmienne lokalne funkcji, którym w chwili jej wywołania przypisano dane reprezentowane przez argumenty. Przy takim założeniu, nazwy parametrów reprezentują jedynie wspomniane zmienne lokalne, spowodowanie zaś, aby operacja na parametrze dotyczyła innej zmiennej wymaga, aby parametr był zmienną wskazującą, a skojarzony z nim argument był wyrażeniem wskazującym zmienną.

```
char Alfa = 'e',
     Beta = 'b',

main()
{
   sub(&Alfa,Beta);
   printf("%c%c", Alfa,Beta);
}

sub(Ref,Val)
   char *Ref,Val;
{
   *Ref = 'j';
   Val = '?';
}
```

W chwili wywołania funkcji sub następuje skojarzenie parametru Ref z argumentem &Alfa i parametru Val z argumentem Beta. Parametrowi Ref zostaje przypisane wskazanie zmiennej Alfa, a parametrowi Val zostaje przypisana dana 'b'. Wykonanie instrukcji

```
*Ref = 'j';
```

powoduje zmianę wartości zmiennej Alfa, natomiast wykonanie instrukcji

```
Val = '?';
```

powoduje zmianę wartości zmiennej Val, ale nie powoduje zmiany wartości zmiennej Beta. Wykonanie programu powoduje wyprowadzenie napisu jb.

Skojarzenie parametru z odpowiadającym mu argumentem jest poprawne tylko wtedy, gdy argument jest zgodny z parametrem. Badanie zgodności odbywa się jednak dopiero po wstępnym przekształceniu argumentów i z uwzględnieniem specjalnego traktowania niektórych parametrów.

Jeśli argument jest typu (char), to zostanie niejawnie przekształcony w argument typu (int) o takiej samej wartości. Jeśli parametr został zadeklarowany jako tablica, to będzie traktowany tak jak zmienna wskazująca dane o postaci elementów tej tablicy. Jeśli parametr został zadeklarowany jako zmienna typu (char), to będzie traktowany tak jakby był zmienną typu (int). Takiej zmiennej zostanie przypisana dana powstała z konwersji przekształconego argumentu do pierwotnego typu parametru.
W sensie przytoczonych ustaleń program

```
char Names[][4] = { "Jan","Ewa" };
main()
{
    output(Names,'B');
}

output(Arr,Chr)
    char Arr[2][4],Chr;
{
    printf("%s%c%s%c", Arr[0],Chr,
                        Arr[1],Chr);
}
```

jest poprawny, ponieważ jest równoważny programowi, w którym funkcja output jest traktowana tak jakby miała postać (w języku wzorcowym)

```
output(Arr,Chr)
    char (*Arr)[4];
    int Chr;
{
    Chr = (char)Chr;
    printf("%s%c  %s%c", Arr[0],Chr,
                         Arr[1],Chr);
}
```

W programie tym, którego wykonanie powoduje wyprowadzenie napisu

JanB EwaB

argument Names typu (char (*)[4]) jest zgodny z przekształconym parametrem Arr, a poddany niejawnej konwersji argument 'B' jest zgodny z parametrem Chr.

W odróżnieniu od funkcji znanych z innych języków programowania, funkcja języka **C** nie musi udostępniać rezultatu. Jeśli jednak wywołanie funkcji występuje w takim kontekście, że posłużenie się rezultatem

jest niezbędne, to jej wykonanie musi zakończyć się wykonaniem instrukcji return zawierającej wyrażenie. Wyrażenie takie, poddane ewentualnej konwersji do typu funkcji stanowi wówczas jej rezultat.

```
char *Ptr = "Jan Bielecki";

main()
{
    char *Name();

    printf("%s", Name());
}

char *
Name()
{
    return Ptr;
}
```

Wykonanie programu powoduje wyprowadzenie napisu Jan Bielecki. W programie tym rezultatem funkcji jest dana wskazująca pierwszy znak tego napisu.

7

WSTĘPNE PRZETWARZANIE PROGRAMU

Integralną część kompilatora systemu HiSoft stanowi *preprocesor* – podsystem umożliwiający wstępne przetwarzanie programu. Otrzymuje on na wejściu ciąg wierszy składających się z dyrektyw preprocesora i dowolnych innych napisów. Dyrektywy preprocesora są odróżniane od napisów tym, że znajdują się w osobnych wierszach i rozpoczynają się od znaku # (*hash*).

Najczęściej używaną dyrektywą preprocesora jest dyrektywa *definiująca*. W ogólnym przypadku ma on postać.

```
#define identyfikator ciąg-jednostek-leksykalnych
```

Zinterpretowanie takiej dyrektywy powoduje związanie z identyfikatorem ciągu-jednostek-leksykalnych i zastępowanie takim ciągiem każdego wystąpienia identyfikatora. Po dokonaniu każdego zastąpienia, otrzymany napis jest analizowany ponownie w celu dokonania ewentualnego następnego zastąpienia. Wymaga się, aby proces zastępowania identyfikatorów ciągami jednostek leksykalnych był skończony.

Dyrektywy definiujące są wykorzystywane najczęściej do związywania mnemonicznych nazw z wybranymi literałami i identyfikatorami programu. W tym sensie następujący program

```
#define NULL 0
#define TRUE 1
#define void int

main()
{
   void out();

   out("Izabela");
}
```

```
void
out(Name)
    char *Name;
{
    if(Name == NULL)
      return;
    while(TRUE){
      printf("%c", *Name);
      if(*Name++ == NULL)
        return;
    }
}
```

jest traktowany tak jak program

```
main()
{
    int out();

    out("Izabela");
}

int
out(Name)
    char *Name;
{
    if(Name == 0)
      return;
    while(1){
      printf("%c", *Name);
      if(*Name++ == 0)
        return;
    }
}
```

Wykonanie tego programu powoduje wyprowadzenie napisu Izabela.

Poza dyrektywami *definiującymi* istotną rolę odgrywają dyrektywy *włączające*. Zostaną one tu omówione w postaci w jakiej występują w systemie HiSoft.

W pierwszym rozdziale tej książki opisano dyrektywę włączającą

```
#include
```

bez parametrów. Jej zinterpretowanie w stanie kompilowania programu powoduje włączenie w miejscu jej wystąpienia tekstu zawartego w bufo-

rze edytora. Dzięki temu jest możliwe wstępne przygotowanie tekstu programu, a następnie skierowanie go do kompilacji.

Inną ważną dyrektywą włączającą jest

#include "*nazwa*"

Zinterpretowanie takiej dyrektywy powoduje włączenie w miejscu jej wystąpienia tekstu zawartego w zbiorze *nazwa*. W przypadku posłużenia się pamięcią kasetową, bezpośrednio po użyciu tej dyrektywy należy włączyć odtwarzanie z magnetofonu i wyłączyć je po odczytaniu zbioru *nazwa*.

W tych przypadkach, gdy zbiór *nazwa* ma charakter biblioteki, z której do programu powinny zostać włączone tylko te funkcje, do których w programie wystąpiły nie zrealizowane jeszcze odwołania, można posłużyć się specjalną dyrektywą włączającą warunkową

#include ?*nazwa*?

Zinterpretowanie takiej dyrektywy powoduje selektywne włączenie zawartości zbioru *nazwa*. W szczególności, jeśli za pomocą edytora przygotowano niekompletny program

```
main()
{
   sub();
   fun();
}

sub()
{
   printf("JanB sub \n");
}
```

a w zbiorze LIBRA znajduje się tekst

```
sub()
{
   printf("Library sub");
}

fun()
{
   printf("Library fun");
}
```

to wykonanie w stanie kompilacji pary dyrektyw

```
#include
#include ?LIBRA?
```

spowoduje utworzenie programu wykonywalnego

```
main()
{
    sub();
    fun();
}

sub()
{
    printf("JanB sub \n");
}

fun()
{
    printf("Library fun");
}
```

w skład którego wchodzi m.in. funkcja fun pochodząca ze zbioru LIBRA.
Uwaga: Spośród omówionych dyrektyw włączających jedynie ta, która zawiera parametr "*name*" należy do definicji języka wzorcowego.

Bardzo przydatną dyrektywą preprocesora jest

#translate *name*

Jest ona wykonywana tuż przed podjęciem kompilowania programu źródłowego, a jej użycie powoduje utworzenie programu nadającego się do samodzielnego, tj. nie wymagającego systemu HiSoft załadowania i wykonania. Kompilowanie po dyrektywie #translate powinno dotyczyć tylko programu poprawnego. Program taki zostanie umieszczony w zbiorze o nazwie name i będzie mógł być załadowany z systemu Basic za pomocą dyrektywy

LOAD "*name*" CODE

i aktywowany za pomocą dyrektywy

RANDOMIZE USR 25200

Uwaga: Dyrektywa #translate nie należy do definicji języka wzorcowego.

8

POSŁUGIWANIE SIĘ BIBLIOTEKAMI FUNKCJI

Kompilator systemu HiSoft dokonuje bezpośredniego przekształcenia *programu źródłowego* w *program wykonywalny*. Z tego powodu uzupełnienie programu podprogramami bibliotecznymi jest możliwe jedynie wtedy, gdy są to podprogramy wbudowane w system, takie jak np. printf, albo podprogramy źródłowe.

Wraz z kompilatorem języka firma HiSoft dostarcza dwa zbiory źródłowe: stdio.h i stdio.lib. Pierwszy z tych zbiorów zawiera szereg użytecznych definicji, m.in.

```
#define NULL 0
#define FALSE 0
#define TRUE -1
#define EOF -1
typedef int FILE;
```

kilka deklaracji wyprzedzających, np.

```
extern unsigned strlen();
```

oraz definicje dwóch rzadko używanych funkcji: max i min.

Jeśli zajdzie potrzeba, wspomniane definicje i deklaracje mogą zostać włączone do programu za pomocą dyrektywy #include. W następującym programie

```
#include "stdio.h"

main()
{
    printf("%d", max('j','b'));
}
```

włączono przed definicję funkcji main wszystkie deklaracje i definicje zawarte w zbiorze stdio.h. Tym samym umożliwiono posłużenie się funk-

Ponieważ posłużenie się *dyrektywą włączenia bezwarunkowego* pociąga za sobą umieszczenie w programie szeregu definicji zbędnych (np. w przytoczonym wcześniej programie, włączenie definicji funkcji min), znacznie częściej włączanie zbiorów bibliotecznych odbywa się za pomocą *dyrektywy włączania warunkowego*. Ilustracją takiej metody postępowania jest program

```
#include "stdio.h"

main()
{
    printf("%d", strlen("jb"));
}

#include ?stdio.lib?
```

w którym druga z dyrektyw włączających została użyta po to, aby ze zbioru stdio.lib dołączyć do programu definicję funkcji strlen.

Użycie w rozpatrzonym programie pierwszej dyrektywy włączającej jest uzasadnione tym, że w zbiorze stdio.h znajduje się deklaracja wyprzedzająca funkcji strlen. Ponieważ wszystkie pozostałe definicje i deklaracje znajdujące się w tym zbiorze są zbędne, można by program uprościć, nadając mu postać

```
extern unsigned strlen();

main()
{
    printf("%d", strlen("jb"));
}

#include ?stdio.lib?
```

uwzględniając fakt, iż ze zbioru stdio.lib zostanie w istocie włączona do programu zaledwie jedna funkcja, można by ją wprost umieścić w programie, nadając mu postać

```
extern unsigned strlen();

main()
{
    printf("%d", strlen("jb"));
}

unsigned
strlen(s)
    char *s;
{
    static unsigned length;
```

```
    length = 0;
    while(*s++) ++length;
    return length;
}
```

Wykonanie tego programu (określającego liczbę znaków ciągu "jb") powoduje wyprowadzenie liczby 2.

Ogółem w zbiorach stdio.h i stdio.lib jest zawartych ponad 40 definicji funkcji. Zapoznanie się z nimi może stanowić ćwiczenie przynoszące wiele korzyści. Krótkie opisy funkcjonalne tych funkcji zamieszczono w Dodatku *C*.

Pozostaje dodać, że nic nie stoi na przeszkodzie, aby użytkownik systemu HiSoft sam tworzył swoje zbiory biblioteczne. W tym celu wystarczy przygotować zestaw deklaracji i definicji za pomocą edytora, a następnie posługując się np. jego dyrektywą

```
p1,9999,JB.LIB
```

umieścić ten zestaw w zbiorze JB.LIB. Należy jedynie pamiętać, aby zgodnie z opisem dyrektywy p podanym w Dodatku *B*, tuż przed naciśnięciem klawisza ENTER kończącego dyrektywę, włączyć magnetofon do zapisu.

9
WYKONYWANIE OPERACJI WEJŚCIA/WYJŚCIA

Operacje wejścia/wyjścia dotyczą *zbiorów danych* albo *urządzeń*. W programach zarówno zbiory danych jak i urządzenia występują pod postacią *plików*. Skojarzenie pliku ze zbiorem albo urządzeniem jest realizowane przez funkcję fopen, a usunięcie tego skojarzenia jest realizowane przez funkcję fclose. Identyfikacja pliku odbywa się za pomocą wskazania plikowego udostępnianego jako rezultat funkcji fopen. Wskazanie to jest daną typu (FILE *) zależnego od implementacji i zdefiniowanego w zbiorze nagłówkowym stdio.h. W systemie HiSoft typ FILE jest zdefiniowany za pomocą deklaracji

```
typedef int FILE;
```

i tym samym nie różni się od typu (int).

W specjalny sposób są traktowane trzy pliki: standardowy plik wejściowy, standardowy plik wyjściowy i standardowy plik do sygnalizowania błędów. W systemie HiSoft standardowy plik wejściowy jest związany z klawiaturą, a standardowy plik wyjściowy i standardowy plik do sygnalizowania błędów jest związany z górną częścią ekranu. Komunikowanie się z klawiaturą, ekranem i drukarką nie wymaga uprzedniego wykonania funkcji fopen, ponieważ z tymi urządzeniami są wstępnie związane ustalone wskazania plikowe. Jeśli przyjąć, że typ (STREAM) jest zdefiniowany jako

```
typedef FILE *STREAM;
```

to obowiązuje następujące przyporządkowanie

Urządzenie	**Wskazanie plikowe**
klawiatura	cast(STREAM)0
górna część ekranu	cast(STREAM)0
dolna część ekranu	cast(STREAM)1
górna część ekranu	cast(STREAM)2
drukarka	cast(STREAM)3

Niektóre funkcje, takie jak np. printf, niejawnie komunikujące się z plikami standardowymi, mogą być wywoływane bez podania wskazania plikowego. Poprawne użycie innych funkcji może wymagać posłużenia się wskazaniem plikowym związanym z plikiem standardowym albo wskazaniem plikowym udostępnionym jako rezultat funkcji fopen.

Funkcja fopen (wbudowana)

Nagłówek: FILE *fopen(name,mode)
char *name,*mode;

Wykonanie funkcji fopen powoduje otwarcie pliku skojarzonego ze zbiorem. Plik zostaje otwarty w trybie określonym przez mode i reprezentuje zbiór o nazwie określonej przez name. Jeśli argumentem skojarzonym z parametrem mode jest "r", to plik zostaje otwarty w trybie do wprowadzenia, a jeśli jest "w", to zostanie otwarty w trybie do wyprowadzania. Rezultatem funkcji jest wskazanie plikowe identyfikujące właśnie otwarty plik. Jeśli otwarcie pliku jest niemożliwe, to rezultatem jest wskazanie puste.

Funkcja fclose (wbudowana)

Nagłówek: fclose(ptr)
FILE *ptr;

Wykonanie funkcji fclose powoduje zamknięcie pliku identyfikowanego przez ptr. Jeśli wspomniany plik był otwarty w trybie do wyprowadzania, to tuż przed jego zamknięciem zostanie uzupełniony znacznikiem końca pliku.

Uwaga: W systemie HiSoft znacznikiem końca pliku jest znak EOF (SYMBOL-SHIFT I). Wykonanie operacji wprowadzania, dotyczącej pliku znajdującego się w pozycji przed znacznikiem, powoduje udostępnienie danej typu (int) o wartości −1 (oznaczanej zazwyczaj identyfikatorem EOF).

```
typedef int FILE;

main()
{
    FILE *out,*fopen();

    out = fopen("JANB","w");
    putc('j',out);
    putc('b',out);
    fclose(out);
}
```

Wykonanie funkcji fopen powoduje otwarcie pliku skojarzonego ze zbiorem JANB w trybie do wyprowadzania. Rezultatem funkcji jest wskazanie plikowe identyfikujące wspomniany plik. Wskazanie to zostaje przypisane zmiennej plikowej out, do której występują odwołania

w wywołaniach funkcji putc i fclose. Tym samym wykonanie tych funkcji dotyczy w istocie zbioru JANB. Wykonanie funkcji putc powoduje wyprowadzenie podanego znaku do pliku określonego przez zmienną plikową out. Wykonanie funkcji fclose powoduje zamknięcie pliku określonego przez tę zmienną, poprzedzone uzupełnieniem go znacznikiem końca pliku. Ostatecznie, wykonanie programu powoduje umieszczenie w zbiorze JANB pary znaków jb oraz znacznika końca pliku.

Funkcja getc (wbudowana)

Nagłówek: int getc(inp)
FILE *inp;

Wykonanie funkcji powoduje wprowadzenie kolejnego znaku z pliku identyfikowanego przez inp. Rezultatem funkcji jest dana typu (int) o wartości równej kodowi wprowadzonego znaku. Jeśli tuż przed wywołaniem funkcji plik znajdował się w pozycji przed znacznikiem końca pliku, to rezultatem jest dana o wartości EOF.

Funkcja ungetc (wbudowana)

Nagłówek: ungetc(c,inp)
char c;
FILE *inp;

Wykonanie funkcji ungetc powoduje cofnięcie do pliku identyfikowanego przez inp, znaku reprezentowanego przez c. Cofnięcie polega na tym, że najbliższe wykonanie funkcji getc spowoduje udostępnienie cofniętego znaku. Jeśli cofnięto pewien znak, to kolejne wywołanie funkcji ungetc musi być poprzedzone udostępnieniem tego znaku.

Funkcja putc (wbudowana)

Nagłówek: int putc(c,out)
char c;
FILE *out;

Wykonanie funkcji putc powoduje wyprowadzenie do pliku identyfikowanego przez out, znaku reprezentowanego przez c. Rezultatem funkcji jest dana typu (int) o wartości równej kodowi wyprowadzonego znaku.

```
typedef int FILE;
typedef FILE *STREAM;
#define EOF (-1)
#define monitor cast(STREAM)0

main()
{
    FILE *inp,*fopen();
    int ch;
```

```
    if(inp = fopen("stdio.h","r")){
      while((ch = getc(inp)) != EOF)
          putc(ch,monitor);
      putc('\n',monitor);
    }
}
```

Wykonanie programu powoduje wyprowadzenie na monitor zawartości zbioru nagłówkowego stdio.h, dostarczanego wraz z kompilatorem.

Funkcja getchar (wbudowana)

Nagłówek: int getchar()

Wykonanie funkcji getchar powoduje wprowadzenie jednego znaku ze standardowego pliku wejściowego. Rezultatem funkcji jest dana typu (int) o wartości równej kodowi wprowadzonego znaku. Jeśli plik znajduje się w pozycji przed znacznikiem końca pliku, to rezultatem jest dana o wartości EOF.

Funkcja putchar (wbudowana)

Nagłówek: int putchar(c)
int c;

Wykonanie funkcji putchar powoduje wyprowadzenie do standardowego pliku wyjściowego, jednego znaku o kodzie c. Rezultatem funkcji jest dana typu (int) o wartości równej kodowi wyprowadzonego znaku.

```
#define EOF (-1)
char flag;

main()
{
    int ch;

    if((ch = getchar()) != EOF)
      main();
    if(!flag){
      flag =1;
      putchar('\n');
    }
    putchar(ch);
}
```

Wykonanie programu powoduje wprowadzenie z klawiatury ciągu znaków zakończonego znakiem EOF (SYMBOL-SHIFT I), a następnie wyprowadzenie tego ciągu w porządku odwrotnym, tzn. od ostatniego znaku do pierwszego.

Funkcja printf (wbudowana)

Nagłówek: printf(fmt, p1, p2, ..., pn)
char *fmt;

Wykonanie funkcji printf powoduje wyprowadzenie do standardowego pliku wyjściowego, pod nadzorem listy wzorców wskazywanej przez fmt, wartości danych reprezentowanych przez p1, p2, ... ,pn. Funkcja printf może być wywoływana ze zmienną liczbą argumentów, o jeden większą niż liczba wzorców konwersji zawartych w ciągu wskazywanym przez fmt. Każdy wzorzec, o ile w ogóle występuje, składa się z ciągu znaków rozpoczynającego się od znaku % (procent), a kończy jedną z liter:

d o x u c s

Znaczenie tych i innych znaków występujących we wzorcach konwersji jest następujące

% rozpoczyna wzorzec konwersji,
– oznacza, że wyprowadzany ciąg znaków ma być wyrównany w polu wyjściowym lewostronnie,
. oddziela liczby występujące we wzorcu,
d powoduje wyprowadzenie argumentu w postaci liczby dziesiętnej,
o powoduje wyprowadzenie argumentu w postaci liczby ósemkowej,
x powoduje wyprowadzenie argumentu w postaci liczby szesnastkowej,
u powoduje wyprowadzenie argumentu w postaci liczby bez znaku,
c powoduje wyprowadzenie argumentu w postaci jednego znaku,
s powoduje wyprowadzenie ciągu znaków wskazywanego przez argument.

Jeśli wzorzec zawiera liczby, np. %6.2s, to ta z nich, która występuje przed kropką określa szerokość pola zewnętrznego. Wyprowadzane znaki są umieszczane w takim polu z wyrównaniem prawostronnym, chyba że bezpośrednio po znaku % wystąpi znak – *(minus)*, kiedy to zostanie zastosowane wyrównanie lewostronne. Druga z liczb, o ile wystąpi, ma znaczenie tylko w połączeniu z konwersją s. Oznacza wówczas liczbę wyprowadzanych znaków ciągu wskazywanego przez argument. Jeśli szerokość pola zewnętrznego nie będzie podana jawnie, to zostanie domniemana jako „dostatecznie duża". Jeśli w ciągu wskazywanym przez fmt występuje znak nie należący do wzorca (być może przedstawiony za pomocą opisu znaku), to jest wyprowadzany dosłownie.

W sensie przytoczonego opisu jest więc słuszne następujące zestawienie

Instrukcja	**Ciąg wyjściowy**
printf("Any text");	Any text
printf("(%–2d)", 5);	(5)

```
printf("(%s)", "JanB");          (JanB)
printf("(%4.3s)", "JanB")        (Jan)
printf("%c", 'B');               B
printf("%x", -1);                FFFF
printf("%d.%d", 1,3);            1,3
```

Funkcja fprintf (wbudowana)

Nagłówek: fprintf(out, fmt, p1, p2, ... ,pn)

```
FILE *out;
char *fmt;
```

Wykonanie funkcji fprintf powoduje wyprowadzenie do pliku identyfikowania przez out, pod nadzorem listy wzorców wskazywanej przez fmt, wartości danych reprezentowanych przez p1, p2, ... ,pn. Funkcja fprintf może być wywołana ze zmienną liczbą argumentów, a jej wykonanie ma analogiczny skutek jak wykonanie funkcji printf. Różnica polega tylko na tym, że ciąg wyprowadzanych znaków może zostać umieszczony w dowolnym pliku, a nie tylko w standardowym pliku wyjściowym.

Funkcja sprintf (wbudowana)

Nagłówek: sprintf(str, fmt, p1, p2, ... ,pn)

```
char *str;
char *fmt;
```

Wykonanie funkcji sprintf powoduje umieszczenie w polu pamięci operacyjnej wskazywanym przez str, pod nadzorem listy wzorców wskazywanej przez fmt, wartości danych reprezentowanych przez p1, p2, ... ,pn. Funkcja sprintf może być wywoływana ze zmienną liczbą argumentów, a jej wykonanie ma analogiczny skutek jak wykonanie funkcji printf. Różnica polega tylko na tym, że ciąg znaków nie jest umieszczany w pliku, lecz we wskazanym obszarze pamięci operacyjnej.

```
#include "stdio.h"
FILE *Inp, *fopen();
char Number[7];
int Ind,Ch,Count;

main()
{
   if(Inp = fopen("JANEK", "r")){
     while((Ch = getc(Inp)) != EOF)
        Count++;
     sprintf(Number, "%u%c", Count, \ 0');
     while(Number[Ind] == ' ')
        Number[Ind ++] = '*';
     printf("JANEK zawiera %s znaków", Number);
   }
}
```

Wykonanie programu powoduje zliczenie znaków zawartych w zbiorze JANEK i wyprowadzenie ich liczby np. w postaci

JANEK zawiera ****13 znaków

Funkcja scanf (wbudowana)

Nagłówek: int scanf(fmt, p1, p2, ... ,pn)
char *fmt;

Wykonanie funkcji scanf powoduje wprowadzenie ze standardowego pliku wejściowego, pod nadzorem listy wzorców wskazywanej przez fmt, ustalonej liczby ciągów znaków, potraktowanie ich jako zapisów wartości danych i przypisanie danych o takich wartościach zmiennym wskazywanym przez p1, p2, ... ,pn. Funkcja scanf może być wywoływana ze zmienną liczbą argumentów, zależną od liczby wzorców konwersji zawartych w ciągu wskazywanym przez fmt. Każdy wzorzec, o ile występuje, składa się z ciągu znaków rozpoczynającego się od znaku % (procent), a kończy jedną z liter

d o x c s

Ciąg znaków wskazywany przez fmt składa się w ogólnym przypadku ze spacji, ze znaków różnych od spacji oraz z wzorców konwersji. Jeśli w takim ciągu *występuje spacja*, to w pliku wejściowym musi odpowiadać jej *ciąg odstępów* (spacji, tabulacji i znaków nowego wiersza). Jeśli w rozpatrywanym ciągu występuje *znak różny od spacji*, to w pliku wejściowym musi występować *dokładnie taki sam znak*. Jeśli natomiast występuje wzorzec konwersji, to określa on rozmiar i sposób interpretowania kolejnego pola wejściowego pliku. Pole takie jest na ogół ograniczone najbliższym odstępem, ale jeśli we wzorcu podano szerokość pola, to liczy ono nie więcej znaków niż to wynika z wzorca. Jeśli bezpośrednio po znaku % występuje znak * *(gwiazdka)*, to zinterpretowanie wzorca powoduje pominięcie najbliższego pola wejściowego pliku.

Zinterpretowanie każdego wzorca (z wyjątkiem wzorca zakończonego literą c) powoduje pominięcie najbliższych odstępów występujących w pliku wejściowym. Po wykonaniu tej czynności następuje zidentyfikowanie pola wejściowego, a następnie zinterpretowanie znaków występujących w tym polu, stosownie do użytego wzorca konwersji. Znaczenie wzorca zakończonego jedną z wymienionych uprzednio liter jest następujące

d w polu wejściowym jest spodziewana liczba dziesiętna,
o w polu wejściowym jest spodziewana liczba ósemkowa,
x w polu wejściowym jest spodziewana liczba szesnastkowa,
c w polu wejściowym jest spodziewany znak,
s w polu wejściowym jest spodziewany ciąg znaków.

Bardzo ważne jest wymaganie, aby argumentami funkcji scanf były wyrażenia wskazujące zmienne o typach zgodnych z użytymi wzorcami konwersji. Na przykład jeśli posłużono się wzorcem %d, to zmienna wskazywana przez argument może być typu (int), ale nie może być typu (char).

Rezultatem funkcji scanf jest dana typu (int). Wartość tej danej określa liczbę zmiennych, którym w następstwie wywołania funkcji przypisano dane utworzone na podstawie pól wejściowych pliku. Jeśli podczas wykonywania funkcji scanf zostanie napotkany znacznik końca pliku, to rezultatem jest dana o wartości EOF.

Funkcja fscanf (wbudowana)

Nagłówek: int fscanf(inp, fmt, p1, p2, ... ,pn)
FILE *inp;
char *fmt;

Wykonanie funkcji fscanf powoduje wprowadzenie z pliku identyfikowanego przez inp, pod nadzorem listy wzorców wskazywanej przez fmt, ustalonej liczby ciągów znaków, potraktowanie ich jako zapisów wartości danych, a następnie przypisanie tych danych zmiennym wskazywanym przez p1,p2, ... ,pn. Funkcja fscanf może być wywoływana ze zmienną liczbą argumentów, a jej wykonanie ma analogiczny skutek jak wykonanie funkcji scanf. Różnica polega tylko na tym, że wprowadzony ciąg znaków może pochodzić z dowolnego pliku, a nie tylko ze standardowego pliku wejściowego.

Funkcja sscanf (wbudowana)

Nagłówek: int sscanf(str,fmt,p1,p2, ... ,pn)
char *str;
char *fmt;

Wykonanie funkcji sscanf powoduje wprowadzenie z pola pamięci operacyjnej, wskazywanego przez str, pod nadzorem listy wzorców wskazywanej przez fmt, ustalonej liczby ciągów znaków, potraktowanie ich jako zapisów wartości danych, a następnie przypisanie tych danych zmiennym wskazywanym przez p1,p2, ... ,pn. Funkcja sscanf może być wywoływana ze zmienną liczbą argumentów, a jej wykonanie ma analogiczny skutek jak wykonanie funkcji scanf. Różnica polega tylko na tym, że wprowadzany ciąg znaków nie pochodzi z pliku, lecz z pola pamięci operacyjnej.

```
int myAge;
char Sep;
char myName[20];
```

```
main()
{
    sscanf("Age=44 JanB=Janek",
           "Age=%d%c %*5s %3s",
           &myAge,&Sep,myName);
    printf("%s is %d%cnow", myName,
                            myAge,
                            Sep);
}
```

Wykonanie programu powoduje wyprowadzenie napisu

Jan is 44 now

Funkcja raw (wbudowana)

Nagłówek: raw()

Wykonanie funkcji raw powoduje wprowadzenie z klawiatury jednego znaku. Dzieje się to bez udziału kursora i bez wyświetlenia wprowadzonego znaku. Rezultatem funkcji jest dana typu (int). Jej wartość jest równa kodowi wprowadzonego znaku.

Funkcja keyhit (wbudowana)

Nagłówek: int keyhit()

Wykonanie funkcji powoduje ustalenie, czy w buforze klawiatury znajduje się nie wprowadzony jeszcze znak. Rezultatem funkcji jest dana typu (int) o wartości 0 *(fałsz)* albo 1 *(prawda)* wyrażająca prawdziwość zdania „w buforze klawiatury znajduje się nie wprowadzony jeszcze znak".

10
ROZSZERZENIA I PRZYKŁADY

Przedstawiony podzbiór języka **C** wystarcza w zupełności do tego, aby konstruować złożone programy do rozwiązywania zagadnień z zakresu programowania systemowego. Tym niemniej, biorąc pod uwagę wyrażoną uprzednio zachętę do samodzielnego zapoznania się z funkcjami zdefiniowanymi w zbiorze nagłówkowym stdio.h i zbiorze bibliotecznym stdio.lib, pokrótce opisanymi w Dodatku **C**, zostaną tu podane informacje dodatkowe na temat języka. Informacje te będą miały pomijalne znaczenie dla tych, którzy nie zamierzają posługiwać się jego rozszerzonymi możliwościami.

Zmienne klasy static i extern

Chociaż nie wyrażono tego jawnie, z *każdą zmienną* programu może być związana jej *klasa*. W uproszczeniu można powiedzieć, że jeśli w deklaracji nie podano klasy, to zmienne zadeklarowane wewnątrz funkcji są klasy *auto*, a zmienne zdefiniowane poza funkcją są klasy *extern*. Różnica pomiędzy tymi dwoma rodzajami zmiennych polega na tym, że zmienne klasy auto są tworzone na nowo tuż przed podjęciem każdego wykonywania funkcji i przestają istnieć po zakończeniu jej wykonywania, natomiast zmienne klasy extern istnieją przez cały czas wykonywania programu. Inną ważną właściwością zmiennych klasy extern jest to, że jeśli w programie występuje kilka deklaracji pewnej zmiennej z atrybutem extern oraz dokładnie jedna deklaracja tej zmiennej bez atrybutu extern, znajdująca się na zewnątrz definicji funkcji, to wszystkie te deklaracje dotyczą tej samej zmiennej.

Równie rzadko jak zmienne klasy extern mogą być używane w programach zmienne klasy *static*. Zmienne takie także istnieją przez cały czas wykonywania programu, ale jeśli zostaną zadeklarowane wew-

W deklaracjach zmiennych klasy static oraz w tych deklaracjach zmiennych klasy extern, w których nie występuje słowo kluczowe extern można umieszczać przypisania danych początkowych.

Funkcje ze zmienną liczbą argumentów

Definicja funkcji, która może być wywoływana ze zmienną liczbą argumentów musi wystąpić w programie przed pierwszym odwołaniem się do takiej funkcji, a w nagłówku definicji bezpośrednio po nawiasie zamykającym listę parametrów, musi wystąpić słowo auto. Jeśli takie warunki zostaną spełnione, to każde wywołanie funkcji zostanie niejawnie uzupełnione argumentem typu (int) określającym ile bajtów zajęły wszystkie argumenty funkcji, z tym argumentem włącznie. Należy nadmienić, że argumenty są wyznaczane w kolejności ich wystąpienia w wywołaniu i są odkładane na stosie rozrastającym się w kierunku adresów niższych. W szczególności rezultatem następującej funkcji max jest maksimum przekazanych jej argumentów typu (int)

```
int
max(Count) auto
    int Count;
{
    int argc, *argv,max;

    argc = (Count >> 1) - 1;
    argv = Count + argc;
    max = -32768;
    while(argc--)
        if(*argv-- > max)
        max = argv[1];
    return max;
}
```

Uwaga: Omówione rozszerzenie nie należy do definicji języka wzorcowego.

Instrukcja inline

Jakość kodu generowanego przez kompilator systemu HiSoft jest na tyle dobra, że potrzeba posłużenia się językiem asemblerowym jest nader rzadka. Tym niemniej w systemie HiSoft istnieje możliwość umieszczania bezpośrednio w programie instrukcji maszynowych. Odbywa się to za pomocą specjalnej instrukcji inline, której argumentami mogą być dowolne wyrażenia, w tym literały szesnastkowe o postaci 0xdd. Przyjmuje się, że każdy argument o wartości z przedziału 0...255 generuje jeden bajt

kodu, a każdy z pozostałych argumentów generuje dwa bajty kodu. W bibliotece stdio.lib instrukcję inline wykorzystano do zdefiniowania podstawowych funkcji graficznych. W szczególności zdefiniowano tam funkcję

```
cls()
{
  inline(0xCD,0xD6B);
}
```

której wykonanie powoduje wyczyszczenie ekranu monitora.
Uwaga: Omówione rozszerzenie nie należy do języka wzorcowego.

Przykłady

1 Zdefiniować funkcję do wyznaczania wartości bezwzględnej argumentu typu (int)

```
int
abs(Par)
   int Par;
{
   return Par < 0 ? -Par : Par;
}
```

2 Zdefiniować funkcję do wyznaczania maksymalnego elementu tablicy o elementach typu (int [ ])

```
int
max(Arr,Len)
   int Len,Arr[ ];
{
   int Val;

   Val = -32768;
   while(Len--)
      if(Arr[Len] > Val)
        Val = Arr[Len];
   return Val;
}
```

3 Zdefiniować funkcję do zliczania bitów 1 w argumencie typu (int)

```
int
count(Bits)
   unsigned Bits;
{
```

```
    for(Tally = 0; Bits; Bits >>= 1)
      Tally += Bits & 1;
    return Tally;
}
```

4 Zdefiniować funkcję, której argumentem jest adres bajtu pamięci operacyjnej, a rezultatem jest dana typu (char) znajdująca się pod tym adresem

```
char
peek(Addr)
  unsigned Addr;
{
  typedef char *ChrPtr;

  return *cast(ChrPtr)Addr;
}
```

5 Zdefiniować funkcję, która pod podanym adresem umieszcza daną typu (char)

```
poke(Addr,Data)
  unsigned Addr;
  char Data;
{
  typedef char *ChrPtr;

  *cast(ChrPtr)Addr = Data;
}
```

6 Zdefiniować funkcję do porównywania ciągów znakowych

```
int
strcmp(Src,Trg)
  char *Src,*Trg;
{
  while(*Src == *Trg++)
    if(!*Src++) return 0;
  return *Src > Trg[-1] ? 1 : -1;
}
```

7 Zdefiniować funkcję do utworzenia danej typu (int) z ciągu znaków składającego się z cyfr

```
int
atoi(Src)
  char *Src;
{
  int Chr,Val;
```

```
    Val = 0;
    while((Chr = *Src++) == ' ')
        Val = 10 * Val + Chr - '0';
    return Val;
}
```

8 Podać program posługujący się funkcjami do dynamicznego zarządzania pamięcią operacyjną (calloc i free) opisanymi w Dodatku *C*. Zrealizować edytor wierszowy umożliwiający kompletowanie tekstu składającego się z wierszy opatrzonych liczbami całkowitymi bez znaku. Przyjąć, że wprowadzenie wiersza wymaga podania liczby i tekstu wiersza. Jeśli tekst wiersza jest pusty, to wiersz jest usuwany. Jeśli nie zostanie podana ani liczba ani tekst, to zostaną wyprowadzone wszystkie wiersze. Przyjąć, że każdy wprowadzany wiersz jest kończony znakiem ENTER, a wprowadzenie z klawiatury znaku EOF (SYMBOL-SHIFT I) powoduje zakończenie wykonywania programu.

```
#include "stdio.h"

typedef struct Str{
            struct Str *next;
            int lab;
          } LIST;
typedef LIST *ListPtr;
typedef char *CharPtr;

LIST head = { NULL };



main()
{
    extern LIST *find();
    int num,del;
    char line[81];
    LIST *pos;

    num = 0;
    do {
      if(!num)
        display();
      else {
        pos = find(num);
        if(pos->next && pos->next->lab == num)
          delete(pos);
        if(del != '\n'){
```

```
            insert(pos,num,line);
          }
        }
        getNumber(&num,&del);
    }   while(del != EOF);
}


LIST *
find(num)
  int(num)
{
  LIST *ref;

  ref = &head;
  while(ref->next && ref->next->lab < num)
      ref = ref->next;
  return ref;
}

delete(pos)
  LIST *pos;
{

  LIST *ptr;

  ptr = pos->next;
  pos->next = ptr->next;
  free(cast(CharPtr)ptr);
}



getLine(ptr,max)
   char *ptr;
   int max;
{
   int i,c;

   for(i = 0;
       i < max - 1 && (c = getchar()) != '\n';
       i++) *ptr++ = c;
   if(c != '\n')
     while(getchar() != '\n');
```

```
    *ptr = '\0';
}

int
length(line)
    char *line;
{
    int len;

    len = 0;
    while(*line++);
        len++;
    return len;
}



insert(pos,num,line)
    LIST *pos;
    int num;
    char line[];
{
    LIST *ptr,*ref;
    int size;

    size = sizeof(ListPtr) + sizeof(int)
                           + length(line) + 1;
    if(ptr = cast(ListPtr)calloc(1,size)){
      ref = pos->next;
      pos->next = ptr;
      ptr->next = ref;
      ptr->lab = num;
      fill(cast(CharPtr)(ptr + 1),line);
    } else
      printf("Zignorowano :- brak pamięci");
}



display()
{
    LIST *ref;

    ref = head.next;
```

```
            printf("%3d %s \n", ref->lab,
                                    cast(CharPtr)(ref + 1));
            ref = ref->next;
        }
    }

fill(ref,line)
  char *ref,*line;
{
  while(*ref++ = *line++);
  *ref = '\0';
}

getNumber(num,del)
   int *num,*del;
{
   int c,val;

   val = 0;
   while((c = getchar()) >= '0' && c <= '9')
      val = 10 * val + c - '0';
   *num = val;
   *del = c;
}

#include ?stdio.lib?
```

9 Podać program, którego wykonanie powoduje wyprowadzenie na ekran monitora zawartości zbioru TEXT.DOC.

```
#define EOF (-1)
typedef int FILE;

main()
{
    int Chr;
    FILE *input,*fopen();

    if((input = fopen("TEXT.DOC","r")) != NULL)
      while((Chr = getc(input)) != EOF)
         putchar(Chr);
}
```

10 Podać program, którego wykonanie powoduje skopiowanie wierszy wprowadzonych z klawiatury na drukarkę

```
typedef int FILE;
#define EOF (-1)
```

```
main()
{
        FILE *STREAM;
        int Chr;

        while((Chr = getchar()) != EOF)
            putc(Chr,cast(STREAM)3);
    }
```

Literatura

1. Bielecki J.: *Język C – interpretacja standardu, WNT 1987*
2. Bielecki J.: *Wprowadzenie do języka C, WNT 1988*
3. Bielecki J.: *Oprogramowanie mikrokomputerów, WKŁ 1987*
4. Kernighan B., Ritchie D.: *Język C, WNT 1987*
5. *HiSoft C for the ZX Spectrum, HiSoft 1984*

Dodatek A – Znaki kodu ASCII

00 nul	20 spacja	40 @	60 '
01 soh	21 !	41 A	61 a
02 stx	22 "	42 B	62 b
03 etx	25 #	43 C	63 c
04 eot	24 $	44 D	64 d
05 enq	25 %	45 E	65 e
06 ack	26 &	46 F	66 f
07 bel	27 '	47 G	67 g
08 bs	28 (	48 H	68 h
09 ht	29)	49 I	69 i
0A lf	2A *	4A J	6A j
0B vt	28 +	4B K	68 k
0C ff	2C ,	4C L	6C l
0D cr	2D −	4D M	6D m
0E so	2E .	4E N	6E n
0F si	2F |	4F 0	6F o
10 dle	30 0	60 P	70 p
11 dc1	31 1	51 Q	71 q
12 dc2	32 2	52 R	72 r
13 dc3	33 3	53 S	73 s
14 dc4	34 4	54 T	74 t
15 nak	35 5	55 U	75 u
16 syn	36 6	56 V	76 v
17 etb	37 7	57 W	77 w
18 can	38 8	58 X	78 x
19 em	39 9	59 Y	79 y
1A sub	3A :	5A Z	7A z
1B esc	3B ;	5B [	7B {
1C fs	3C <	5C \	7C |
1D gs	3D =	5D]	7D }
1E rs	3E >	5E ^	7E ~
1F us	3F ?	5F _	7F del

Dodatek B – Edytor systemu HiSoft

Dyrektywy edytora mają postać ogólną

Dn1,n2,t1,t2

w której D jest jednoliterową nazwą dyrektywy, n1 i n2 są liczbami całkowitymi z przedziału 1...32767, a t1 i t2 są napisami z których każdy liczy nie więcej niż 20 znaków. W wielu przypadkach wymienione tu liczby i napisy mogą być opuszczone. Jeśli są one niezbędne, ale nie zostały podane jawnie i nie mogą być domniemane, to otrzymują takie wartości jak w dyrektywie użytej uprzednio.

Jeśli system HiSoft znajduje się w stanie kompilowania programu, to wywołanie edytora wymaga naciśnięcia klawisza EDIT (CAPS-SHIFT 1), a po nim klawisza ENTER. Wywołanie edytora zostanie wówczas potwierdzone pojawieniem się na ekranie znaku zachęty >*(większe)*. Zakończenie edycji i wywołanie kompilatora systemu HiSoft odbywa się za pomocą dyrektywy C, a wywołanie systemu Basic za pomocą dyrektywy B. Żadna z tych dyrektyw nie wymaga parametrów.

Wiersze tekstu przygotowywanego za pomocą edytora są opatrywane liczbami całkowitymi. Powoływanie się na te liczby umożliwia wyrywkowe operowanie na już wprowadzonym tekście. Wspomniane liczby nie są zapamiętywane w pamięci zewnętrznej i są ignorowane przez kompilator. Mają one znaczenie jedynie w procesie edycji tekstu.

Litery identyfikujące dyrektywy mogą być przedstawione jako duże albo małe. Jeśli zostanie użyta dyrektywa nieistniejąca, albo gdy zapis dyrektywy okaże się niepoprawny, to dyrektywa zostanie zignorowana. Niekiedy na ekranie pojawi się wówczas napis Pardon?

Najprostszy, chociaż nie najwygodniejszy, sposób wprowadzenia tekstu polega na niezależnym wprowadzaniu jego wierszy. Wprowadzenie wiersza uzyskuje się przez wprowadzenie z klawiatury liczby opatrującej wiersz, spacji i tekstu wiersza, zakończonego znakiem ENTER. Jeśli tekst wiersza nie mieści się w wierszu ekranu, to jest automatycznie kontynuowany w wierszach następnych. Wymaga się jedynie, aby rozmiar wiersza tekstu nie przekroczył 80 znaków. Jeśli podczas wprowadzania tekstu zostanie popełniony błąd, to można go natychmiast usunąć, naciskając klawisz DELETE (CAPS-SHIFT 0). Kilkakrotne naciśnięcie tego klawisza eliminuje więcej znaków tekstu. Bardziej złożone zmiany w tekście można realizować za pomocą opisanych dalej dyrektyw i poddyrektyw.

Dyrektywa I (Insert)

Zapis dyrektywy: Im,n

Wykonanie dyrektywy I powoduje zainicjowanie automatycznego wpro-

ekranie pojawia się liczba m i rozpoczyna się kompletowanie wiersza opatrzonego tą liczbą. Po wprowadzeniu znaku ENTER następuje zakończenie kompletowania wiersza m, a na ekranie pojawia się liczba m+n opatrująca następny wiersz, a po kolejnych znakach ENTER liczby m+2n, m+3n itd. W ten sposób następuje skompletowanie wierszy opatrzonych liczbami m+i*n. Jeśli po wprowadzeniu pewnej z takich liczb zostanie naciśnięty klawisz EDIT (CAPS-SHIFT 1), to wykonywanie dyrektywy I zostanie zakończone i na ekranie ponownie pojawi się znak zachęty. Należy nadmienić, że jeśli podczas wykonywania dyrektywy I zostanie skompletowany wiersz opatrzony taką samą liczbą jak wiersz wprowadzony uprzednio (za pomocą innej dyrektywy I albo bezpośrednio), to nastąpi usunięcie wiersza wprowadzonego wcześniej i zastąpienie go wierszem wprowadzonym później.

Dyrektywa L (List)

Zapis dyrektywy: Lm,n

Wykonanie dyrektywy L powoduje wyprowadzenie na ekran wierszy opatrzonych liczbami z przedziału m...n. Jeśli w dyrektywie zostanie opuszczony argument m, to będzie domniemane m=1, a jeśli zostanie opuszczony argument n, to będzie domniemane n=32767. Wynika stąd w szczególności, że użycie dyrektywy L bez argumentów spowoduje wprowadzenie na ekran całego dotychczas skompletowanego tekstu.

Jeśli wprowadzany tekst nie mieści się na ekranie w całości, to jest wprowadzany porcjami. Liczba wierszy wprowadzanych w porcji wynosi 10, ale może być zmieniona za pomocą dyrektywy K. Jeśli po wprowadzeniu porcji wierszy zostanie naciśnięty klawisz EDIT (CAPS-SHIFT 1), to dalsze wprowadzenie tekstu zostanie zaniechane. Naciśnięcie dowolnego innego klawisza spowoduje wprowadzenie kolejnej porcji tekstu.

Dyrektywa K (Chunk)

Zapis dyrektywy: Km

Wykonanie dyrektywy K z argumentem m powoduje, że podczas wykonywania dyrektywy L tekst zawarty w buforze edytora będzie wyprowadzany porcjami po m wierszy.

Dyrektywa W (Write)

Zapis dyrektywy: Wm,n

Wykonanie dyrektywy W powoduje wyprowadzenie na drukarkę wierszy opatrzonych liczbami z przedziału m...n . Jeśli w dyrektywie zostanie opuszczony argument m, to zostanie domniemane m=1, a jeśli zostanie opuszczony argument n, to zostanie domniemane n=32767. Wynika stąd w szczególności, że użycie dyrektywy W bez argumentów spowoduje wprowadzenie na drukarkę całego dotychczas skompletowanego tekstu.

Jeśli podczas wprowadzenia tekstu zostanie naciśnięty klawisz BREAK (CAPS-SHIFT spacja), to dalsze wprowadzenie tekstu zostanie zaniechane.

Dyrektywa S (Set)

Zapis dyrektywy: S,,d

Wykonanie dyrektywy S powoduje zmianę separatora argumentów dyrektyw. Początkowo tym separatorem jest , *(przecinek)*, ale może on być zamieniony na dowolny inny znak różny od spacji. Po takiej zmianie, nowy separator musi być używany we wszystkich następnych dyrektywach, włącznie z dyrektywą S.

Dyrektywa V (View)

Zapis dyrektywy: V

Wykonanie dyrektywy V powoduje ujawnienie argumentów n1, n2, t1 i t2, jakie będą użyte w najbliższej dyrektywie, w której nie zostały podane jawnie i nie mogą być domniemane. Ponadto wykonanie dyrektywy V spowoduje wyprowadzenie adresu początku i końca bufora, w którym został skompletowany dotychczas wprowadzony tekst.

Dyrektywa D (Delete)

Zapis dyrektywy: Dm,n

Wykonanie dyrektywy D powoduje usunięcie wierszy tekstu opatrzonych liczbami z przedziału m...n . W przypadku gdy usunięciu ma podlegać tylko jeden wiersz, należy przyjąć m=n. Znacznie łatwiej jest uzyskać ten skutek wprowadzając z klawiatury jedynie liczbę m, a następnie naciskając klawisz ENTER. Ponieważ w dyrektywie D nie są stosowane domniemania, oba jej argumenty muszą być podane jawnie.

Dyrektywa M (Move)

Zapis dyrektywy: Mm,n

Wykonanie dyrektywy M ma taki skutek jak wykonanie dyrektywy Dn,n a następnie wprowadzenie wiersza opatrzonego liczbą n identycznego z wierszem opatrzonym liczbą m. Wynika stąd, że dyrektywa M może być użyta do powielania wierszy.

Dyrektywa N (Renumber)

Zapis dyrektywy: Nm,n

Wykonanie dyrektywy N powoduje usunięcie wszystkich liczb opatrujących wiersze tekstu, a następnie opatrzenie wierszy wziętych w ich pierwotnej kolejności, liczbami m, m+n, m+2n itd., tj. przenumerowanie wszystkich wierszy. Ponieważ w dyrektywie N nie są stosowane domnie-

Dyrektywa P (Put)

Zapis dyrektywy: Pm,n,t

Wykonanie dyrektywy P powoduje zapamiętanie w pamięci zewnętrznej, jako zbioru o nazwie t, tych wierszy tekstu, które są opatrzone liczbami z przedziału m...n . Nazwa zbioru w stacji microdrive musi być poprzedzona numerem stacji i oddzielona od niego znakiem : *(dwukropek)*. Jeśli wykonanie dyrektywy P dotyczy pamięci kasetowej, to dyrektywa może być użyta dopiero po włączeniu magnetofonu do zapisu.

Dyrektywa G (Get)

Zapis dyrektywy: G,,t

Wykonanie dyrektywy G powoduje wprowadzenie z pamięci zewnętrznej tekstu zawartego w zbiorze o nazwie t. Nazwa zbioru w stacji microdrive musi być poprzedzona numerem stacji i oddzielona od niego znakiem : *(dwukropek)*. Wprowadzony tekst zostanie umieszczony bezpośrednio za dotychczas skompletowanym tekstem. Jego poszczególne wiersze będą automatycznie opatrzone liczbami całkowitymi z przyrostem 10. Podobnie jak w dyrektywie P, nazwa zbioru w stacji microdrive musi być poprzedzona numerem stacji i oddzielona od niego znakiem : *(dwukropek)*.

Dyrektywa C (Compile)

Zapis dyrektywy: C

Wykonanie dyrektywy powoduje wywołanie kompilatora systemu HiSoft.

Dyrektywa B (Basic)

Zapis dyrektywy: B

Wykonanie dyrektywy B powoduje wywołanie systemu Basic. Po wykonaniu prostych czynności w tym systemie, takich jak np. zmiana koloru i tła znaków, można ponownie wywołać system HiSoft. W tym celu należy wykonać instrukcję

RANDOMIZE USR 25200

Dyrektywa F (Find)

Zapis dyrektywy: Fm,n,f,s

Wykonanie dyrektywy F powoduje odszukanie wśród wierszy opatrzonych liczbami z przedziału m...n, takiego wiersza opatrzonego najmniejszą liczbą, który zawiera ciąg znaków f. Po znalezieniu wspomnianego wiersza nastąpi niejawne wykonanie opisanej dalej dyrektywy E dla tego właśnie wiersza i usytuowanie kursora pod pierwszym znakiem tekstu f. Tak odszukany tekst f może być następnie zmieniony na tekst s. W tym celu należy posłużyć się opisaną dalej poddyrektywą S.

Dyrektywa E (Edit)

Zapis dyrektywy: Em

Wykonanie dyrektywy E może być jawne albo niejawne. Wykonanie jawne ma miejsce wtedy, gdy system HiSoft znajduje się w stanie edycji, a z klawiatury zostanie wprowadzona dyrektywa E. Wykonanie niejawne może nastąpić podczas kompilowania programu, którego pewien wiersz m zawiera błąd składniowy. W takiej sytuacji na ekran jest wyprowadzany komunikat identyfikujący błąd, a po naciśnięciu klawisza EDIT (CAPS-SHIFT 1) jest niejawnie wykonywana dyrektywa Em. W przypadku naciśnięcia dowolnego innego klawisza jest wywoływany kompilator.

Niezależnie od sposobu wykonania dyrektywy E następuje zainicjowanie edycji wiersza opatrzonego liczbą m. Jeśli wiersz opatrzony taką liczbą nie istnieje (co może mieć miejsce tylko po wywołaniu jawnym), wykonywanie dyrektywy jest uznawane za zakończone. W przeciwnym razie na ekran jest wyprowadzana liczba m, po niej tekst wiersza opatrzonego taką liczbą, a w nowym wierszu ponownie liczba m oraz kursor mający postać litery L.

Po wykonaniu tych czynności, edytor znajduje się w stanie umożliwiającym dokonanie edycji wiersza wyprowadzonego na ekran. Edycja ta jest realizowana za pomocą jednoznakowych poddyrektyw, które mogą dotyczyć zarówno wiersza w jego początkowej postaci – nazywanego tu *źródłowym*, jak i wiersza tworzonego na skutek edycji – nazywanego *wynikowym*.

poddyrektywa =>

Wykonanie tej poddyrektywy wymaga naciśnięcia klawisza => (CAPS-SHIFT 8). Jeśli w chwili użycia tej dyrektywy jest wyświetlony cały wiersz wynikowy, to następuje przeniesienie do niego kolejnego znaku wiersza źródłowego i przemieszczenie kursora o jedną pozycję w prawo. W przeciwnym razie następuje wyświetlenie następnego znaku wiersza wynikowego i przesunięcie kursora o jedną pozycję w prawo.

poddyrektywa <=

Wykonanie tej poddyrektywy wymaga naciśnięcia klawisza <= (CAPS-SHIFT 5). Po wykonaniu tej czynności następuje wygaszenie ostatniego wyświetlanego znaku wiersza wynikowego i przesunięcie kursora o jedną pozycję w lewo.

poddyrektywa ENTER

Wykonanie tej poddyrektywy wymaga naciśnięcia klawisza ENTER. Spowoduje to przeniesienie do wiersza wynikowego pozostałych zna-

ście kompletowanym przez edytor, wiersza źródłowego – wierszem wynikowym.

poddyrektywa Q (Quit)

Wykonanie tej poddyrektywy wymaga naciśnięcia klawisza Q. Spowoduje to zakończenie edycji wiersza, bez uwzględnienia ewentualnie dokonanych w nim zmian.

poddyrektywa R (Replace)

Wykonanie tej poddyrektywy wymaga naciśnięcia klawisza R. Spowoduje to usunięcie wszystkich znaków wiersza wynikowego i będzie miało taki skutek jakby edycję wiersza źródłowego rozpoczynano od nowa.

poddyrektywa L (List)

Wykonanie tej poddyrektywy wymaga naciśnięcia klawisza L. Spowoduje to przeniesienie do wiersza wynikowego pozostałych znaków wiersza źródłowego i przemieszczenie kursora na początek wiersza wynikowego. Ma to taki skutek, że wiersz wynikowy staje się wierszem źródłowym, a nowy wiersz wynikowy staje się wierszem pustym.

poddyrektywa K (Kill)

Wykonanie tej poddyrektywy wymaga naciśnięcia klawisza K. Jeśli jest wyświetlony cały wiersz wynikowy, to spowoduje to zignorowanie najbliższego znaku wiersza źródłowego. W przeciwnym razie nastąpi usunięcie z wiersza wynikowego tego znaku, który jest pod kursorem.

poddyrektywa Z (Zeroize)

Wykonanie tej poddyrektywy wymaga naciśnięcia klawisza Z. Spowoduje to pominięcie wszystkich znaków wiersza wynikowego, począwszy od znaku ukrytego pod kursorem oraz wszystkich następnych – jeszcze nie rozpatrywanych – znaków wiersza źródłowego.

poddyrektywa I (Insert)

Wykonanie tej poddyrektywy wymaga naciśnięcia klawisza I. Spowoduje to zmianę kursora ze znaku L na znak * *(gwiazdka)* i przejście do trybu, w którym znaki wprowadzane z klawiatury wchodzą bezpośrednio do wiersza wynikowego. W tym trybie pomyłki mogą być natychmiast usuwane za pomocą klawisza DELETE (CAPS-SHIFT 0). Bezpośrednie wprowadzanie znaków kończy się z chwilą naciśnięcia klawisza ENTER. Powrót do zwykłego trybu interpretowania poddyrektyw wymaga naciśnięcia klawisza ENTER.

poddyrektywa X (External)

Wykonanie tej poddyrektywy wymaga naciśnięcia klawisza X. Spowoduje to przeniesienie do wiersza wynikowego pozostałych znaków wier-

sza źródłowego, a następnie wyświetlenie całego wiersza wynikowego i niejawne wykonanie poddyrektywy I.

poddyrektywa C (Change)

Wykonanie tej poddyrektywy wymaga naciśnięcia klawisza C. Spowoduje to zmianę kursora ze znaku L na znak + *(plus)* i przejście do trybu, w którym znaki wprowadzane z klawiatury wchodzą bezpośrednio do wiersza wynikowego. Jeśli w chwili wprowadzania takiego znaku jest wyświetlony cały wiersz wynikowy, to znak jest umieszczany na końcu wiersza wynikowego, a kolejny znak wiersza źródłowego jest pomijany. W przeciwnym razie wprowadzany znak zastępuje ten znak wiersza wynikowego, który jest ukryty pod kursorem, a bezpośrednio po tym jest niejawnie wykonywana poddyrektywa =>. Powrót do zwykłego trybu interpretowania poddyrektyw wymaga naciśnięcia klawisza ENTER.

poddyrektywa F (Find)

Wykonanie tej poddyrektywy wymaga naciśnięcia klawisza F. Powinno to nastąpić po niejawnym wykonaniu dyrektywy E spowodowanym wykonaniem dyrektywy F. Spowoduje to odszukanie następnego wystąpienia tekstu f wymienionego w uprzednio wykonanej dyrektywie F. Po odszukaniu wiersza zawierającego wspomniany tekst zostanie niejawnie wywołana jego edycja, a kursor będzie usytuowany na pozycji pierwszego znaku odszukanego tekstu. Należy nadmienić, że poszukiwanie tekstu f będzie ograniczone do zakresu określonego w dyrektywie F.

poddyrektywa S (Substitute)

Wykonanie tej poddyrektywy wymaga naciśnięcia klawisza S. Spowoduje to zastąpienie odszukanego tekstu f, tekstem s określonym w dyrektywie F, a następnie niejawne wykonanie poddyrektywy F. Należy nadmienić, że odszukanie tekstu powinno być zainicjowane wykonaniem dyrektywy albo poddyrektywy F.

Dodatek C – Wybrane funkcje biblioteczne

Arytmetyka, operacje znakowe i konwersje

Funkcja max (stdio.h)

Nagłówek: int max(count) auto
int count;

Wykonanie funkcji powoduje udostępnienie danej typu (int) o wartości równej maksimum z jej argumentów.

Funkcja min (stdio.h)

Nagłówek: int min(count) auto
int count;

Wykonanie funkcji powoduje udostępnienie danej typu (int) o wartości równej minimum z jej argumentów.

Funkcja abs (stdio.lib)

Nagłówek: int abs(p)
int p;

Wykonanie funkcji powoduje udostępnienie danej typu (int) o wartości równej modułowi jej argumentu.

Funkcja sign (stdio.h)

Nagłówek: int sign(p)
int p;

Wykonanie funkcji powoduje udostępnienie danej typu (int) o wartości −1, 0 albo 1, stosownie do znaku argumentu.

Funkcja peek (stdio.lib)

Nagłówek: char peek(addr)
unsigned addr;

Wykonanie funkcji powoduje udostępnienie danej typu (char) reprezentowanej w tym bajcie pamięci, który ma adres addr.

Funkcja poke (stdio.lib)

Nagłówek: poke(addr,ch)
unsigned addr;
int ch;

Wykonanie funkcji powoduje umieszczenie w bajcie pamięci o adresie addr, znaku reprezentowanego przez ch.

Funkcja tolower (wbudowana)

Nagłówek: char tolower(c)
char c;

Wykonanie funkcji powoduje udostępnienie danej typu (char). Jeśli c reprezentuje literę dużą, to dana ta reprezentuje odpowiadającą jej literę małą. W pozostałych przypadkach reprezentuje taki sam znak jak c.

Funkcja toupper (wbudowana)

Nagłówek: char toupper(c)
char c;

Wykonanie funkcji powoduje udostępnienie danej typu (char). Jeśli c reprezentuje literę małą, to dana ta reprezentuje odpowiadającą jej literę dużą. W pozostałych przypadkach reprezentuje taki sam znak jak c.

Funkcja atoi (stdio.lib)

Nagłówek: int atoi(str)
char *str;

Wykonanie funkcji powoduje konwersję ciągu znaków wskazywanego przez str na daną typu (int). Jeśli ciąg rozpoczyna się od liczby całkowitej, to wartością tej danej jest wartość liczby. W przeciwnym razie wartością danej jest 0.

Sortowanie

Funkcja qsort (stdio.lib)

Nagłówek: qsort(ptr,cnt,size,fun)
char *ptr;
int cnt,size;
int (*fun)();

Wykonanie funkcji powoduje posortowanie danych tworzących tablicę o cnt elementach, której pierwszy bajt jest wskazywany przez ptr. Rozmiar elementu jest określony przez size, a funkcja porównująca przez fun. Funkcja skojarzona z parametrem fun musi być tak dobrana, aby dla elementów elm1 i elm2 rezultatem wywołania

(*fun)(&elm1,&elm2)

była dana typu (int) o wartości −1, 0 albo 1, zgodnie z następującym zestawieniem

−1 jeśli elm1 < elm2
0 jeśli elm1 = elm2
+1 jeśli elm1 > elm2

Przetwarzanie ciągów znakowych

Funkcja strcat (stdio.lib)

Nagłówek: char *strcat(first,second)

Wykonanie funkcji powoduje wydłużenie ciągu znaków wskazywanego przez first, o ciąg znaków wskazywany przez second. Rezultatem funkcji jest dana typu (char *) wskazująca pierwszy znak ciągu first.

Funkcja strcmp (stdio.lib)

Nagłówek: int strcmp(str,trg)
char *src,*trg:,

Wykonanie funkcji powoduje porównanie ciągów wskazywanych przez src i trg i udostępnienie danej typu (int) określającej wynik porównania. Dana ta ma wartość 0 – jeśli ciągi są identyczne, wartość dodatnią – jeśli pierwszy ciąg jest większy niż drugi albo wartość ujemną – jeśli pierwszy ciąg jest mniejszy niż drugi. Porównywanie odbywa się do końca ciągów albo do stwierdzenia nieidentyczności kolejnej pary porównywanych znaków. W tym drugim przypadku ciągi są różne, a większy jest ten, którego znak ma większy kod.

Funkcja strcpy (stdio.lib)

Nagłówek: char *strcpy(dst,src)
char *dst,*src;

Wykonanie funkcji powoduje skopiowanie ciągu znaków wskazywanego przez src, do obszaru pamięci wskazywanego przez dst i udostępnienie wskazania dst.

Funkcja strlen (stdio.lib)

Nagłówek: unsigned strlen(str)
char *str;

Wykonanie funkcji powoduje udostępnienie danej typu (int) o wartości równej liczbie znaków ciągu wskazywanego przez str.

Zarządzanie pamięcią operacyjną

Funkcja calloc (stdio.lib)

Nagłówek: char *calloc(n,size)
unsigned n,size;

Wykonanie funkcji powoduje przydzielenie obszaru pamięci operacyjnej wystarczającego do pomieszczenia n obiektów o rozmiarze size każdy i udostępnienie wskazania na tak przydzielony obszar. Jeśli przydzielenie obszaru jest niemożliwe, to będzie udostępnione wskazanie puste (NULL).

Funkcja free (stdio.lib)

Nagłówek: free(ptr)
char *ptr;

Wykonanie funkcji powoduje zwolnienie obszaru pamięci operacyjnej przydzielonego za pomocą funkcji calloc. Argument funkcji reprezentuje wskazanie udostępnione podczas wykonywania funkcji calloc.

Funkcja swap (wbudowana)

Nagłówek: swap(pArea,qArea,len)
```
char *pArea,*qArea;
unsigned len;
```

Wykonanie funkcji powoduje zamianę miejscami zawartości dwóch obszarów pamięci operacyjnej o długości len bajtów każdy, wskazywanych przez pArea i qArea.

Funkcja move (wbudowana)

Nagłówek: move(dst,src,len)
```
char *dst,*src;
unsigned len;
```

Wykonanie funkcji powoduje skopiowanie obszaru pamięci operacyjnej o długości len, wskazywanego przez src, do obszaru wskazywanego przez dst.

Wykonywanie operacji wejścia/wyjścia

Wszystkie udostępnione w systemie HiSoft wbudowane funkcje wejścia/wyjścia zostały opisane w rozdz. 9. W tym miejscu zostaną przedstawione jedynie funkcje dodatkowe zdefiniowane w bibliotece stdio.lib.

Funkcja gets (stdio.lib)

Nagłówek: char *gets(str)
```
char *str;
```

Wykonanie funkcji powoduje umieszczenie w polu pamięci operacyjnej wskazywanym przez str, ciągu znaków pochodzących ze standardowego pliku wejściowego. Umieszczanie znaków kończy się z chwilą wprowadzenia znaku '\n' (ENTER). Zamiast tego znaku w polu pamięci jest umieszczany znak o kodzie 0. Rezultatem jest wskazanie pierwszego znaku pola pamięci.

Funkcja puts (stdio.lib)

Nagłówek: puts(str)
```
char *str;
```

Wykonanie funkcji powoduje wyprowadzenie do standardowego pliku wyjściowego, ciągu znaków wskazywanego przez str. Wymaga się, aby ostatnim znakiem tego ciągu był znak o kodzie 0.

Funkcja fgets (stdio.lib)

Nagłówek: char *fgets(str,n,inp)
char *str;
int n;
FILE *inp;

Wykonanie funkcji powoduje umieszczenie w polu pamięci operacyjnej wskazywanym przez str, nie więcej niż n znaków pochodzących z pliku identyfikowanego przez inp. Umieszczanie znaków w polu kończy się z chwilą wprowadzenia n−1 znaków, albo po wyprowadzeniu znaku '\n'. Wówczas w polu jest dodatkowo umieszczany znak o kodzie 0. Rezultatem funkcji jest wskazanie pierwszego znaku pola, chyba że podczas wykonywania funkcji został napotkany znacznik końca pliku. W takim przypadku rezultatem jest wskazanie puste.

Funkcja fputs (stdio.lib)

Nagłówek: fputs(str,out)
char *str;
FILE *out;

Wykonanie funkcji powoduje wyprowadzenie do pliku identyfikowanego przez out, ciągu znaków wskazywanego przez str, zakończonego znakiem o kodzie 0.

Generowanie liczb losowych

Funkcja rand (stdio.lib)

Nagłówek: int rand()

Rezultatem funkcji jest dana typu (int) o wartości przypadkowej.

Funkcja srand (stdio.lib)

Nagłówek: srand(s)
int s;

Wykonanie funkcji powoduje wykonanie nowego "posiewu" generatora liczb losowych.

Grafika i dźwięk

Funkcja plot (stdio.lib)

Nagłówek: plot(on,x,y)
int on,x,y;

Wykonanie funkcji powoduje wyświetlenie piksela o współrzędnych (x,y) w kolorze papieru (dla on=0) albo atramentu (dla on=1).

Funkcja line (stdio.lib)

Nagłówek: line(on,dx,dy)
int on,dx,dy;

Wykonanie funkcji powoduje wykreślenie odcinka linii prostej od punktu bieżącego (określonego np. przez plot) do punktu oddalonego od niego o dx w poziomie i dy w pionie. Punkty odcinka są wykreślane w kolorze papieru (dla on=0) albo atramentu (dla on=1).

Funkcja paper (stdio.lib)

Nagłówek: paper(color)
int color;

Wykonanie funkcji powoduje zinterpretowanie argumentu jako numeru koloru papieru (0 – *czarny*, 1 – *niebieski*, 2 – *czerwony*, 3 – *karmazynowy*, 4 – *zielony*, 5 – *turkusowy*, 6 – *żółty*, 7 – *biały)* i dokonanie odpowiedniej zmiany koloru papieru.

Funkcja ink (stdio.lib)

Nagłówek: ink(color)
int color;

Wykonanie funkcji powoduje zinterpretowanie argumentu jako numeru koloru atramentu (0 – *czarny*, 1 – *niebieski*, 2 – *czerwony*, 3 – *karmazynowy*, 4 – *zielony*, 5 – *turkusowy*, 6 – *żółty*, 7 – *biały)* i dokonanie odpowiedniej zmiany koloru atramentu.

Funkcja cls (stdio.lib)

Nagłówek: cls()

Wykonanie funkcji powoduje wyczyszczenie ekranu.

Funkcja beep (stdio.lib)

Nagłówek: beep(time,tone)
int time,tone;

Wykonanie funkcji powoduje wygenerowanie dźwięku o częstotliwości tone Hz, utrzymywanego przez time/10 s.

Dodatek D – Kody błędów

ERROR – *0* – missing 'x'
brak spodziewanego znaku
ERROR – *1* – RESTRICTION: not implemented
nieznany typ danych
ERROR – *2* – bad character constant
błąd w literale znakowym
ERROR – *3* – not a preprocesor command
błędna dyrektywa preprocesora
ERROR – *4* – LIMIT: macro buffer full
definicja odwołuje się do samej siebie
ERROR – *5* – can only define identifiers as macros
błąd w dyrektywie preprocesora
ERROR – *6* – RESTRICTION: macros may not have parameters
błąd w dyrektywie preprocesora
ERROR – *7* – cannot open file
brak zbioru o podanej nazwie
ERROR – *8* – RESTRICTION: cannot nest includes
włączony zbiór zawiera dyrektywę włączającą
ERROR – *9* – missing while
źle skonstruowana instrukcja do
ERROR – *10* – not in loop or switch
instrukcja break w złym kontekście
ERROR – *11* – not in loop
instrukcja continue w złym kontekście
ERROR – *12* – not in switch
instrukcja case albo default w złym kontekście
ERROR – *13* – LIMIT: too many case statements
zbyt wiele przedrostków case
ERROR – *14* – multiple default statements
więcej jak jeden przedrostek default
ERROR – *15* – goto needs a label
brak etykiety w instrukcji goto
ERROR – *16* – multiple use of identifier
kolizja zmiennej i identyfikatora
ERROR – *17* – direct execution not possible
niewłaściwe użycie dyrektywy preprocesora
ERROR – *18* – LIMIT: name table full
zbyt wiele typów danych
ERROR – *19* – LIMIT: too many types
zbyt wiele typów danych
ERROR – *20* – duplicate declaration – type
podwójna deklaracja

ERROR – *21* – duplicate declaration – storage class
podwójna deklaracja
ERROR – *22* – LIMIT: global symbol table full
zbyt wiele zmiennych na zewnątrz funkcji
ERROR – *23* – LIMIT: to much global data
zbyt wiele zmiennych na zewnątrz funkcji
ERROR – *24* – duplicate declaration
podwójna deklaracja
ERROR – *25* – LIMIT: local symbol table full
zbyt wiele zmiennych w funkcji
ERROR – *26* – this variable was not in parametr list
zmienna nie była wymieniona w liście parametrów
ERROR – *27* – undefined variable
program zawiera nie zrealizowane odwołania
ERROR – *28* – bad function return type
błędne wyrażenie w instrukcji return
ERROR – *29* – no arrays of functions
posłużono się tablicą funkcji
ERROR – *30* – LIMIT: expression too complicated
zbyt skomplikowane wyrażenie
ERROR – *31* – LIMIT: expression too complicated
zbyt skomplikowane wyrażenie
ERROR – *32* – bad type combination
niewłaściwy argument operacji
ERROR – *33* – bad operand type
niewłaściwy argument operacji
ERROR – *34* – need an lvalue
nie użyto l-wyrażenia
ERROR – *35* – not a defined member of a structure
nazwa nie identyfikuje pola
ERROR – *36* – expected a primary here
nie użyto wyrażenia pierwotnego
ERROR – *37* – undefined variable
odwołano się do zmiennej nie zadeklarowanej
ERROR – *38* – need a type name
nie użyto nazwy typu
ERROR – *39* – need a constant expression
nie użyto wyrażenia stałego
ERROR – *40* – can only call functions
w wywołaniu nie użyto nazwy funkcji
ERROR – *41* – : does not follow ? properly
źle użyty operator trójargumentowy
ERROR – *42* – destination must be an lvalue
nie użyto l-wyrażenia
ERROR – *43* – need a : to follow ?
źle użyty operator trójargumentowy

ERROR – *44* – need a pointer
nie użyto wskazania
ERROR – *45* – illegal parameter type
niewłaściwy parametr funkcji
ERROR – *46* – RESTRICTION: not implemented
nieznany typ danych
ERROR – *47* – cannot use this operator
nieznany typ danych
ERROR – *48* – bad declaration
błąd w deklaracji
ERROR – *49* – storage class not valid
błąd w deklaracji
ERROR – *50*
(zarezerwowany)
ERROR – *51* – duplicate declaration of structure tag
podwójna deklaracja oznacznika
ERROR – *52* – use a predeclared structure for parameters
zadeklaruj typ strukturowy poza funkcją
ERROR – *53* – structure cannot contain itself
struktura nie może zawierać samej siebie
ERROR – *54* – bad declarator
błąd w deklaratorze
ERROR – *55* – missing) in function declaration
brak nawiasu w deklaracji funkcji
ERROR – *56* – bad formal parameter list
błąd w deklaratorze funkcji
ERROR – *57* – type should be function
nieznany typ rezultatu funkcji
ERROR – *58*
– *59*
(zarezerwowany)
ERROR – *60* – LIMIT: no more memory
brak pamięci
ERROR – *61* – RESTRICTION: use assignment to initialize
niedozwolone przypisanie danej początkowej
ERROR – *62* – cannot initialize this
niedozwolone przypisanie danej początkowej
ERROR – *63* – cannot initialize this
niedozwolone przypisanie danej początkowej
ERROR – *64* – too much initialization data
zbyt wiele danych początkowych

Dodatek E – Mikrokomputer Elwro 800 Jr

Mikrokomputer Elwro 800 Jr został opracowany do celów edukacyjnych. W jednym z jego trybów pracy zachowuje się tak jak mikrokomputer ZX Spectrum i z tego powodu tekst niniejszej książki stosuje się w równym stopniu do każdego z wymienionych mikrokomputerów.

Użytkownicy Elwro 800 Jr powinni jedynie sięgnąć do tomu dokumentacji pt. *Podręcznik użytkownika* ... w celu uświadomienia sobie odwzorowania klawiatury Elwro na klawiaturę ZX Spectrum. Z lektury tej wyniknie w szczególności następujące przyporządkowanie klawiszy

ZX Spectrum	**Elwro 800 Jr**
CAPS SHIFT	SHIFT
SYMBOL SHIFT	ALT
DELETE	BS
EDIT	ESC
ENTER	CR

Literatura

Wojciech Cellary, Jarogniew Rykowski: *Podręcznik użytkownika mikrokomputera Elwro 800 Jr* – Dodatek D.

Wydawnictwa Komunikacji i Łączności
Warszawa 1988
Wydanie I. Nakład 49 650 + 350 egz.
Ark. wyd. 5. Ark. druk. 6 (7,98 A)
Oddano do składania w listopadzie 1987
Podpisano do druku w lutym 1988
Papier offset III kl. 70 g, rola 70 cm
Zamówienie P/103/87. K/9941
Prasowe Zakłady Graficzne RSW „Prasa-Książka-Ruch"
w Łodzi, ul. Armii Czerwonej 28 Z.3110/87 U-54

Cena zł 250.–

3095