INFORMATYKA
KOMPUTERY
SYSTEMY

zeszyty programów komputerowych nr 2/88

Dodatek „Żołnierza Wolności" Cena 200 zł

JĘZYK „C"

CZĘŚĆ 1

NOWOCZESNY KOMPUTER NA !

GRY

DEEP BLUE C

SPIS TRESCI

PODRĘCZNIK PROGRAMOWANIA

I

OPIS KOMPILATORA JĘZYKA C

dla mikrokomputerów ATARI XL/XE

Język C staje się coraz powszechniej stosowanym narzędziem programowania. Posiada on wiele zalet, takich jak: szczególna przydatność do tworzenia oprogramowania systemowego, możliwość definiowania i przetwarzania nowoczesnych struktur danych oraz prostota i zwartość budowanych programów. Inną jego zaletą, być może najistotniejszą, jest to, że nie jest on związany z konkretnym typem sprzętu i napisane w nim programy mogą być uruchamiane na dowolnym komputerze, dla którego opracowano kompilator tego języka. Intensywny wzrost liczby mikrokomputerów, użytkowanych zarówno przez amatorów jak i poważnych użytkowników, znacznie przybliża możliwość programowania w tym języku. Warto zatem zainteresować się nim bliżej.

Kierując się tym, prezentujemy publikowane po raz pierwszy na łamach popularnych czasopism informatycznych, kompleksowe opracowanie poświęcone językowi C. Składa się ono z dwóch części. Niniejsza, pierwsza część przedstawia pakiet kompilacyjny DEEP BLUE C języka C opracowany dla mikrokomputerów ATARI serii XL/XE. Omówiono w niej zasady posługiwania się programami użytkowymi tego pakietu oraz scharakteryzowano biblioteki funkcji standardowych. Część ta zawiera również obszerny podręcznik programowania, przedstawiający podstawowe elementy i konstrukcje języka ilustrowane licznymi programami przykładowymi. Część druga, która zostanie opublikowana w następnym zeszycie specjalnym IKS-a poświęcona będzie omówieniu obszernej biblioteki funkcji, realizujących arytmetykę zmiennoprzecinkową w kompilatorze DEEP BLUE C oraz zasad ich wykorzystania w tworzonych programach.

Z. Dyjak

H. Krasuski

1. WSTĘP

Język C - skąd taka dziwna nazwa na określenie języka programowania? Skąd język ten się wywodzi i czym różni się od innych języków programowania? Co za jego pomocą można zrobić lepiej niż przy zastosowaniu innych języków?

Tego typu pytania nurtują każdego, kto po raz pierwszy zetknie się z językiem C. Pytania te są jeszcze bardziej intrygujące, gdy uwzględni się fakt, że prawie każdy użytkownik mikrokomputera pisał użyteczne programy w najbardziej popularnym języku, jakim jest BASIC, a być może również w innych językach wysokopoziomowych jak PASCAL.

Jednak język C uderza każdego swą odmiennością od tych języków.

C jest językiem programowania ogólnego zastosowania, zaprojektowanym w celu wypełnienia luki pomiędzy językiem BASIC a językiem asemblera. Stworzony przez programistów systemowych jest wysokopoziomową alternatywą dla języków asemblerowych. Język C jest szybszy i oferuje programiście więcej możliwości niż BASIC, a jednocześnie jest mniej błędotwórczy i bardziej wydajny niż język asemblera.

Kilka słów o historii powstania języka C. Jego twórcami są Dennis M. Ritchie i Brian W. Kernighan. Powstał on w latach siedemdziesiątych jako kolejna modyfikacja języka BCPL, który opracowano na Uniwersytecie w Cambridge. BCPL jest do dzisiaj używany na tej uczelni do nauczania programowania. Język C jest bardzo silnie związany z systemem UNIX.

Pierwsza wersja systemu operacyjnego UNIX została napisana w języku asemblera. W efekcie UNIX mógł być eksploatowany tylko na komputerach określonego typu. Aby uciec od ścisłego przywiązania systemu do określonego typu komputera opracowano specyficzny język, nazwany "B", przeznaczony do przenoszenia systemu UNIX z jednego typu komputera na inny.

Z języka B poddanego pewnym modyfikacjom, dokonanym przez Dennisa M. Ritchie, powstał język C. Ritchie dokonał również przeprogramowania systemu UNIX, stosując nowoopracowany język. W efekcie UNIX może być eksploatowany na dowolnym typie komputera. Również programy napisane w języku C mogą być uruchamiane na dowolnym komputerze, dla którego opracowano kompilator tego języka.

Jednym z kompilatorów C opracowanych dla mikrokomputerów ATARI serii XL/XE jest kompilator DEEP BLUE C. Umożliwia on tworzenie programów o różnym przeznaczeniu (w tym programów graficznych, muzycznych itp.) i różnej wielkości, w tym programów dużych, na które składa się więcej niż sto linii kodu źródłowego.

Programy napisane w języku C dla tych mikrokomputerów, mimo iż działają nieco wolniej niż programy w kodzie maszynowym, to są znacznie łatwiejsze w tworzeniu i testowaniu. Takie elementy języka jak: wskaźniki, funkcje rekurencyjne i wysokopoziomowe struktury sterujące czynią nawet potencjalnie skomplikowany program łatwym do zakodowania i przetestowania. Co więcej, C jest faktycznie językiem oprogramowania systemowego nowej generacji komputerów; programy w nim napisane, w przeciwieństwie do języka asemblera, będą mogły zostać przeniesione na inne komputery (również te, które nie są oparte na mikroprocesorze 6502) kosztem nieznacznych modyfikacji.

2. KOMPILATOR DEEP BLUE C

2.1. Wymagane akcesoria sprzętowo - programowe.

Do poprawnej pracy kompilatora DEEP BLUE C potrzebne są:

- 48 kB pamięci RAM;
- stacja dysków ATARI;
- edytor tekstowy.

Ponadto dla pełnego wykorzystania możliwości oferowanych przez kompilator DEEP BLUE C /DBC/, szczególnie do włączania do programu tworzonego w języku C, procedur w kodzie maszynowym, może okazać się przydatnym ATARI Macro Assembler (MAC65) lub pakiet ATARI Assembler Editor.

2.2. Różnice w stosunku do "C" standardowego.

Należy stwierdzić, że język C nie posiada formalnego protokółu, aczkolwiek prace nad jego powstaniem prowadzone są od dłuższego czasu przez ANSI. Jednak we wszystkich publikacjach na jego temat za wersję standardową uznaje się wersję języka zdefiniowaną przez Kernighana i Ritchie'a /K&R/ w pracy "The C programming language" (wydano zostało polskie tłumaczenie tej książki). Niektórzy autorzy za standardowe uważają również pewne rozszerzenia języka C, powstałe przy tworzeniu systemu UNIX.

Z oczywistych powodów kompilator DBC nie posiada zaimplementowanej pełnej wersji języka C, zdefiniowanej przez K&R. W szczególności DBC nie zapewnia realizacji:

1/ struktur i unii;

2/ tablic wielowymiarowych;

3/ liczb zmiennoprzecinkowych;

4/ funkcji zwracających inne wartości niż typu int;

5/ operatora unarnego "sizeof";

6/ operatora binarnego konwersji typów "typecasting".

Ponadto DBC posiada następujące niestandardowe cechy:

1/ ostatni punkt instrukcji "switch", "case" lub "default" musi być zakończony instrukcją "break", "continue" lub "return";

2/ usunięto starą konstrukcję =<op>. Zamiast niej należy używać <op>=;

3/ znaki (zmienne typu char) traktowane są jako liczby

bez znaku. Przyjmują one wartości od 0 do 255;

4/ łańcuchy znaków nie mogą być wydłużone do następnej linii logicznej;

5/ długość jednej linii kodu źródłowego nie może przekroczyć 79 znaków;

6/ funkcje mogą mieć maksymalnie 126 argumentów.

DBC zapewnia realizację następujących elementów standardowego "C":

1/ typy danych: char, int, pointer;

2/ tablice jednowymiarowe;

3/ operatory unarne (jednoargumentowe): *, -, &, %, ++, —, !(wykrzyknik), ~(tylda);

4/ operatory binarne (dwuargumentowe): *, -, &, /, %, |(pionowa kreska), ^, &, ==, !=, <, >, <=, >=, <<, >>, <op>=, &&, ::, przecinek;

5/ trójargumentowy operator warunkowy ?

6/ instrukcje (słowa kluczowe) języka C: if, else, while, break, continue, return, for, do, switch, case, default;

7/ dyrektywy kompilatora: #define i #include;

8/ relokacyjny linker;

9/ stałe: szesnastkowe, ósemkowe, i typu "backslash".

Standardowy język C wykorzystuje kilka znaków ASCII niedostępnych w komputerach ATARI; i tak nawiasy klamrowe "{,}" zostały zastąpiono kombinacjami znaków $(i $), a znak tyldy (~) zastąpiono parą znaków "$-". Ponieważ znak dolara "$" nie jest wykorzystywany w standardowym C, to można przy pomocy funkcji edytora "znajdź i podaleń" zamienić programy napisane w standardowym C na format akceptowany przez DBC.

2.3. Dyskietka zawierająca kompilator DEEP BLUE C.

Przed rozpoczęciem poznawania zasad programowania w języku C warto jest zapoznać się z zawartością dyskietki, na której zapisany jest kompilator DBC, poznać mechanizm kompilacji oraz nauczyć się posługiwania programami realizującymi ten proces.

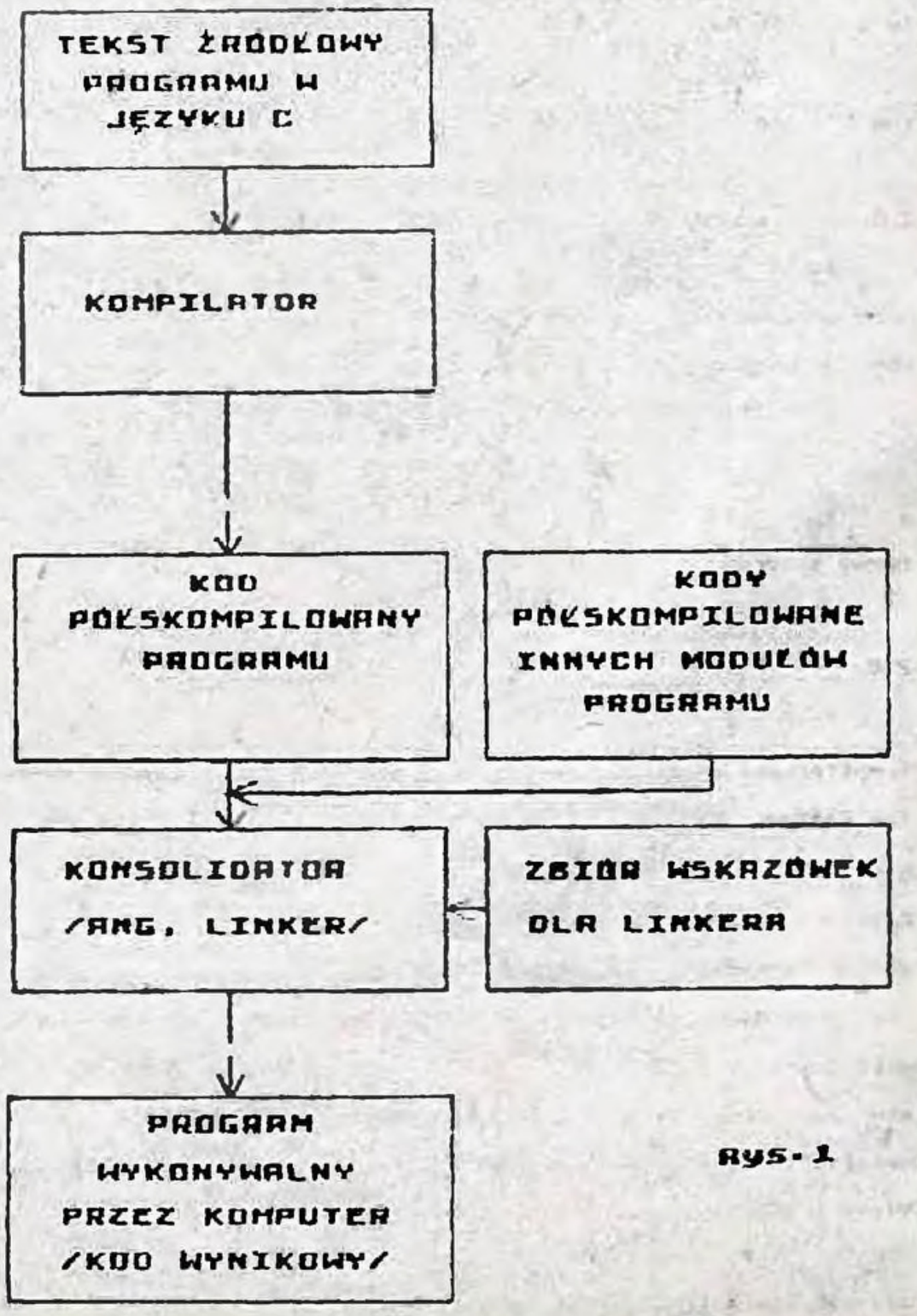
Dyskietka zawierająca kompilator zapisana jest dwustronnie. Na stronie pierwszej zawarte są niżej wymienione zbiory:

1. DOS.SYS - standardowy FMS DOS'a;
2. DUP.SYS - standardowy zbiór zawierający menu DOS'a;
3. CC.COM - kompilator DBC;
4. CLINK.COM - linker DBC;
5. DBC.OBJ - moduł uruchomieniowy DBC;
6. AIO.C - tekst źródłowy funkcji biblioteki we/wy;
7. AIO.CCC - postać półskompilowana funkcji biblioteki AIO;
8. GRAPHICS.C - tekst źródłowy funkcji graficznych i we/wy dla gier;
9. GRAPHICS.CCC - postać półskompilowana biblioteki GRAPHICS.C;
10. PMG.C - tekst źródłowy funkcji realizujących grafikę PMG;
11. PMG.CCC - postać półskompilowana biblioteki PMG;
12. PRINTF.C - tekst źródłowy funkcji wyprowadzania sformatowanego;
13. PRINTF.CCC - postać półskompilowana biblioteki PRINTF;
14. X.C - tekst źródłowy programu demonstracyjnego (patrz opis w dalszej części);
15. X.CCC - postać półskompilowana programu X;
16. X.LNK - zbiór wskazówek dla linkera, wykorzystywany w procesie konsolidacji (linkowania) programu X;
17. X.COM - wykonywalna wersja programu X;
18. BOUNCE.C - tekst źródłowy, postać półskompilowana, itd. programu demonstrującego wykorzystanie funkcji graficznych.

Druga strona dyskietki zawiera bibliotekę funkcji realizujących arytmetykę zmiennoprzecinkową oraz programy demonstrujące sposób ich wykorzystania. Przed rozpoczęciem pracy z kompilatorem należy wykonać roboczą kopię tej dyskietki.

2.4. Proces kompilacji i konsolidacji.

Pisanie programu jest przekazywaniem komputerowi sposobu (algorytmu) postępowania, niezbędnego do rozwiązania określonego problemu, przy czym forma przekazu jest ściśle określona przez język programowania. Język C, podobnie jak każdy inny język programowania posiada ściśle określone reguły syntaktyczne (gramatyczne) i semantyczne (znaczeniowe) - zbliżone do reguł rządzących naturalnymi językami, którymi porozumiewają się ludzie. Program napisany w języku C nie może być "zrozumiały" przez komputer w formie takiej, w jakiej został napisany - postać ta nazywana będzie dalej kodem źródłowym programu. Musi on być przetłumaczony przez odpowiedni program do postaci, którą komputer "zrozumie" i wykona. Taki program tłumaczący nazywany jest kompilatorem. Kompilator DBC tłumaczy kod źródłowy programu na postać przejściową (jeszcze nie wykonywalną przez komputer, a potrzebną do przeprowadzenia pełnego procesu kompilacji), zwaną kodem lub programem półskompilowanym. Aby program półskompilowany był wykonywalny przez komputer, musi być najpierw połączony z kodami półskompilowanymi procedur użytkowych, stanowiących część kompilatora, zwaną w języku C - biblioteką kompilatora. Dopiero w wyniku połączenia powstaje kod programu, który może być wykonany przez komputer. Ten



Rys. 1

wykonywalny kod nazywany jest kodem wynikowym. Proces łączenia realizowany jest przez odpowiedni program, będący również integralną częścią kompilatora, zwany linkerem. Odpowiednikami nazw "linker" i "linkowanie" są w języku polskim nazwy "konsolidator" i "proces konsolidacji". Proces konsolidacji, dla programisty przyzwyczajonego do takich języków programowania jak PASCAL czy BASIC, może początkowo wydać się niezrozumiały i niepotrzebny. I chociaż jest on dodatkowym przebiegiem w procesie kompilacji programu w języku C, to właśnie dzięki niemu język ten ma wiele zalet w porównaniu z innymi językami programowania. Praktyką stosowaną powszechnie przez programistów piszących programy w "C" jest tworzenie programu "po kawałku", oddzielne kompilowanie każdego fragmentu programu i na końcu łączenie takich "kawałków" w jeden program. Każdy napisany, skompilowany i przetestowany fragment jakiegoś programu może być w postaci półskompilowanej dołączony do biblioteki kompilatora i wykorzystany w innym programie, bez konieczności tworzenia go od początku. Dzięki takiej technice proces tworzenia programów w języku C jest bardzo wygodny i znacznie szybszy niż w innych językach wysokopoziomowych. Proces kompilacji programu w języku C jest przedstawiony na rys. 1.

Reasumując: aby przekształcić kod źródłowy programu w języku C, napisany na papierze, na wykonywalny przez komputer kod wynikowy, należy wykonać niżej przedstawione czynności:

1. Przy pomocy edytora tekstowego przygotować program jako jeden lub kilka zbiorów z tekstem źródłowym;

2. Przekształcić zbiory źródłowe w zbiory z kodem półskompilowanym poprzez skompilowanie tych pierwszych przy pomocy programu CC.COM kompilatora DBC;

3. Stworzyć zbiór wskazówek dla linkera. Zbiór ten powinien zawierać nazwy wszystkich zbiorów z kodem półskompilowanym, które są częściami programu i jest wykorzystywany przez linker (w kroku 4) do połączenia razem wszystkich części programu;

4. Przy pomocy linkera CLINK.COM połączyć wszystkie zbiory, zawierające kod półskompilowany w kompletny program wynikowy.

2.5. Rozszerzenia nazw zbiorów.

Każda nazwa zbioru, wykorzystywanego w wyżej opisanym przebiegu musi mieć swoje własne rozszerzenie, wymagane przez kompilator DBC. Rozszerzenia te są następujące:

.C - zbiór zawierający tekst źródłowy programu;

.CCC - zbiór zawierający postać półskompilowaną programu;

.LNK - zbiór zawierający informacje dla linkera, wskazujące nazwy zbiorów, które mają być łączone;

.COM - zbiór zawierający wykonywalny kod wynikowy programu.

2.6. Tworzenie zbiorów z tekstem źródłowym programu.

Zbiory z tekstem źródłowym programu (zbiory źródłowe dla kompilatora) najlepiej przygotowuje się przy pomocy edytora tekstowego. Do edytowania programów w języku C doskonale nadają się: ATARI Program Text Editor (dołączony do Macro Assemblera), edytor kompilatora KYAN PASCAL lub każdy inny edytor tekstów, jak ATARI WRITER czy Speedscript. Każdy inny, nie wymieniony wyżej edytor tekstowy można z powodzeniem wykorzystać do edycji programów w języku C. Ważne jest, aby stosowany edytor nie wstawiał kolejnego numeru linii na początku każdego wiersza. Nie jest więc możliwe wykorzystanie do edycji edytora BASIC-a ani edytora z pakietu ATARI Assembler Editor, jeżeli nie napiszemy specjalnego programu użytkowego, usuwającego z tekstu źródłowego programu w języku C numery wierszy. Warto jeszcze raz podkreślić, że wszystkie zbiory źródłowe powinny mieć rozszerzenie ".C". Rozszerzenie to przyjęto tradycyjnie i jest ono wymagane przez kompilator DBC, jak również przez inne kompilatory języka C opracowane dla różnorodnych mikrokomputerów, również profesjonalnych.

2.7. Kompilacja zbiorów źródłowych.

Jak zaznaczono, program źródłowy w języku C, aby mógł być uruchomiony na komputerze musi zostać "przetłumaczony" na kod wynikowy. Proces "tłumaczenia" składa się z dwóch etapów: kompilacji i łączenia. Kompilacja wykonywana jest przez program o nazwie CC.COM kompilatora DBC. Czyta on instrukcje programu źródłowego ze zbioru źródłowego (rozszerzenie .C), tłumaczy je na postać półskompilowaną i zapisuje je do tworzonego zbioru z kodem półskompilowanym (rozszerzenie .CCC). Przedstawimy teraz dokładny sposób posługiwania się tym programem. Na przykład: aby skompilować program źródłowy zapisany w zbiorze o nazwie X.C należy wykonać następujące czynności:

1. Włączyć stację dysków i włożyć do niej dyskietkę z kompilatorem DBC;

2. Włączyć komputer przytrzymując wciśnięty klawisz OPTION;

3. Gdy wyświetli się menu DOS-a wybrać opcję L (ładowanie zbioru), napisać CC.COM i wcisnąć RETURN;

4. Gdy kompilator CC.COM załaduje się do pamięci komputera, wyczyści ekran i wyświetli komunikat nagłówkowy:

```
Deep Blue C Compiler version 1.2
(C) 1982 John Howard Palevich
File to compile (or RETURN to exit)
(zbiór do kompilacji (lub RETURN - wyjście z kompilatora));
```

5. Należy włożyć do stacji dysków dyskietkę zawierającą zbiór z programem źródłowym, napisać nazwę tego zbioru i nacisnąć RETURN. O ile pełna nazwa zbioru ma postać, na przykład D:X.C to wystarczy podać tylko główną część tej nazwy, w tym przypadku X, ponieważ kompilator zakłada, że pozostała część nazwy składa się z "D:" i nazwa ma rozszerzenie ".C". Wystarczy zatem napisać tylko X i nacisnąć RETURN. Jeżeli dyskietka z programem źródłowym znajduje się w stacji o numerze innym niż 1, to obowiązkowo należy podać w nazwie, poprzedzony literą D numer stacji dysków, zawierającej dyskietkę z programem do kompilacji, np. "D2:X".

6. Kompilator wypisze komunikat:

```
D:X.C-->D:X.CCC
```

który oznacza, że rozpoczęła się kompilacja programu. Kompilacja może trwać kilka minut, w zależności od długości i stopnia skomplikowania programu źródłowego. Aby poinformować programistę o tym, co się aktualnie dzieje, kompilator wyświetla nazwę funkcji z programu źródłowego, którą w danej chwili analizuje. Jeżeli w programie źródłowym znalezione zostaną błędy syntaktyczne, kompilator wyświetla wiersz kodu źródłowego, w którym wykrył błąd a wyświetlona pod nim strzałka pionowa skierowana ku górze wskazuje dokładne miejsce jego wystąpienia, np:

```
main()${printf("Hello,world/n");
        ↑
Missing final end.
```

(brak nawiasu zamykającego "}" funkcję main())

Jeżeli błędy syntaktyczne nie występują, kompilator wyświetli komunikat no errors (nie ma błędów) i znów pojawi się komunikat nagłówkowy:

File to compile (or RETURN to exit)

Jeżeli za jednym razem chcemy skompilować więcej niż jeden program, możemy podać nową nazwę zbioru z programem źródłowym i ponownie uruchomić kompilator, naciskając RETURN.

7. Gdy chcemy zakończyć kompilowanie zbiorów źródłowych naciskamy klawisz RETURN (bez podawania nazwy zbioru) aby powrócić do menu Dos-a.

2.8. Łączenie zbiorów półskompilowanych.

Gdy poszczególne zbiory źródłowe zostaną już skompilowane bez błędów, to należy je połączyć i stworzyć kod wynikowy programu. W tym celu należy najpierw utworzyć zbiór informacji dla linkera, wskazujący zbiory z kodem półskompilowanym, które mają być połączone. Zbiór wskazówek dla linkera musi mieć rozszerzenie ".LNK". W zbiorze tym, oprócz nazw zbiorów zawierających kod półskompilowany powstały po kompilacji zbiorów źródłowych, muszą być podane również nazwy bibliotek kompilatora DBC, których funkcje wykorzystywane są przez program źródłowy. Typowy, mały program, wyprowadzający na ekran monitora dowolny tekst przy pomocy funkcji wyprowadzania sformatowanego "printf()" wymagałby zbioru wskazówek dla linkera o niżej przedstawionej postaci:

WYSWIETL
PRINTF
DBC.OBJ

gdzie: "WYSWIETL" jest nazwą zbioru zawierającego kod półskompilowany tego programu a "DBC.OBJ" jest nazwą programu uruchomieniowego kompilatora DBC, który musi być dołączony do kodu wynikowego każdego programu. Jeżeli dyskietka zawierająca wyżej wyszczególnione zbiory znajduje się w stacji o innym numerze niż jeden, to podane nazwy muszą posiadać prefiks "Dn:", gdzie n jest numerem stacji dysków. Jeżeli rozszerzenia nie są podane, to linker przyjmuje rozszerzenia ".CCC". Zbiór wskazówek dla linkera nie może zawierać linii pustych, nawet na końcu zbioru.

Po utworzeniu zbioru wskazówek dla linkera można przystąpić do konsolidacji wszystkich zbiorów półskompilowanych w kod wynikowy programu. W tym celu należy załadować do pamięci komputera DOS i wybrać opcję L z jego menu. Następnie należy załadować linker, pisząc CLINK.COM i naciskając RETURN. Linker DBC po załadowaniu wyświetli komunikat:

Deep Blue C linker version 1.2
(C) 1982 John Howard Palevich
Link program, Duplicate file or Quit

(łączenie, kopiowanie zbioru lub wyjście z linkera)

Linker uruchamia się podając pierwszą literę komendy (L,D lub Q). Funkcja L tworzy działający program w języku C. Funkcja D pozwala na kopiowanie małych zbiorów (o pojemności do ok 6kB) z jednej dyskietki na drugą, bez potrzeby wykorzystania w tym celu opcji O z DOS-a. Na dużych zbiorach funkcja ta nie działa. Funkcja Q powoduje powrót do DOS-a. Zatem, gdy chcemy wykonać proces konsolidacji piszemy:

L X i naciskamy RETURN;

gdzie X jest nazwą zbioru zawierającego wskazówki dla linkera (linker przyjmuje w tym przypadku, że nazwa tego zbioru ma rozszerzenie ".LNK"). Podobnie uruchamia się funkcję kopiowania zbiorów dostępną w linkerze.

Linker nie wykona łączenia jeżeli określone zbiory nie istnieją na dyskietce. Dodatkowo, jeśli w trakcie łączenia linker napotka funkcję lub zmienną zewnętrzną, która nigdzie nie została zdefiniowana, to wyświetli komunikat: undefined label (niezdefiniowaną etykietę) i listę niezdefiniowanych nazw funkcji i zmiennych. Jeżeli np.: źle napisano nazwę zmiennej ("alfa"), która była zdefiniowana z nazwą "alpha", to również zostanie to zasygnalizowane. Jeśli nie ma błędów, linker wyświetli komunikat no errors i ponawia zaproszenie do dalszej pracy "Link program, Duplicate file or Quit". Po zakończeniu łączenia programu można wyjść z linkera pisząc Q i wciskając RETURN.

2.9. Uruchomienie programu wynikowego.

Skompilowany i skonsolidowany program w języku C może być traktowany jak każdy program w kodzie maszynowym. Można go nawet opatrzyć nazwą AUTORUN.SYS aby po włączeniu komputera ładował i uruchamiał się automatycznie. Podobnie jak inne zbiory z programami można go również ładować za pomocą opcji L z DOS-a. Podczas działania takiego programu mogą wystąpić cztery rodzaje błędów, spowodowanych złą konstrukcją programu. Błędy te nazywane są błędami uruchomieniowymi. Gdy jeden z tych błędów wystąpi, program zatrzymuje się i wyświetlany jest komunikat:

dbc i run-time error <litera>
type a key to return to DOS

<litera> może być następująca:

A - stos przekroczył zakres zmiennej RAMTOP. Zdarza się to, gdy w uruchomionym programie występuje nieskończona rekurencja lub gdy zdefiniowano zbyt wiele zmiennych.

B - nielegalny kod operacji. Błąd ten ma miejsce wówczas, gdy program wychodzi z obszaru kodu i próbuje wykonać jako rozkaz, zawartość komórek zawierających "śmieci".

C - błąd wersji. Ma miejsce wówczas, gdy numery wersji programów CC.COM, CLINK.COM i DBC.OBJ są różne.

D - próba dzielenia przez zero. W uruchomionym programie wykryta została operacja dzielenia przez zero.

2.10. Programy demonstracyjne.

Na dyskietce zawierającej pakiet kompilacyjny DBC znajdują się trzy programy demonstracyjne, które pomagają zapoznać się z programami pakietu i z zasadami ich wykorzystania. Dla każdego programu demonstracyjnego podany jest jego kod źródłowy, postać półskompilowana, zbiór wskazówek dla linkera wykorzystywany w procesie konsolidacji tego programu oraz kod wynikowy.

Pierwszy program o nazwie X demonstruje niektóre możliwości wprowadzania i wyprowadzania informacji w DBC na/z urządzenia zewnętrzne jakz stacja dysków, monitor i klawiatura. Aby uruchomić ten program, należy załadować DOS-a a następnie opcją L załadować program X.COM. Program ten uruchomi się automatycznie i wyświetli komunikat:

File reader/writer
Command: r, w, q

Przy pomocy tego programu można odczytać zbiór ze stacji dysków (przez naciśnięcie r), zapisać go (komenda w) lub zakończyć pracę (komenda q). Dobrym ćwiczeniem jest przeanalizowanie postaci źródłowej tego programu, zawartej w zbiorze X.C, ilustrującej sposób użycia funkcji z biblioteki kompilatora DBC, takich jak: open(), close(), gets(), getchar() i printf().

Drugi program demonstruje wykorzystanie grafiki PMG. Zawiera przykłady użycia takich funkcji bibliotecznych jak: color(), plot(), drawto(), sound(), pmcinit(), pmmove() i pmcolor(). Program ten zapisany jest w zbiorze o nazwie BOUNCE.COM a uruchamia się go tak, jak program X.COM.

Przejrzenie jego postaci źródłowej jest również dobrym ćwiczeniem.

Ostatni program demonstruje wykorzystanie biblioteki matematycznej kompilatora DBC, której funkcje służą do realizacji obliczeń w arytmetyce zmiennoprzecinkowej.

Opis poszczególnych bibliotek kompilatora DBC, funkcji w nich zawartych i zasad ich użycia w programie przedstawiony jest w dodatkach, umieszczonych na końcu zeszytu.

2.11. Uwagi techniczne.

Analizując uważnie postać źródłową funkcji wejścia/wyjścia zawartych w bibliotece AIO.C można zauważyć, że niektóre funkcje takie jak: ciov() zdefiniowane są w szczególny sposób, z wykorzystaniem instrukcji asm. Na przykład:

```
ciov(iocb,com,bad,blen,ax1,ax2)
int iocb,com,blen,ax1,ax2;
char *bad;
asm 12291
```

W powyższy sposób definiuje się podprogramy napisane w kodzie maszynowym. Gdy wywołana zostanie funkcja "asm" to jej argumenty przekazywane są na stos, podobnie jak w funkcji USR() z języka BASIC i następuje skok pod adres następujący po słowie "asm". Podobnie, jeżeli chcemy wykonać skok do własnego podprogramu w kodzie maszynowym, który zaczyna się od adresu $600 wystarczy w programie w języku C umieścić poniższą sekwencję instrukcji:

```
foo()
asm 600
```

Jest rzeczą oczywistą, że nasz podprogram musi zostać najpierw dołączony do programu wynikowego w języku C. W tym celu musimy wytworzyć kod maszynowy tego podprogramu za pomocą dowolnego asemblera i zapisać go, jako zbiór binarny na dyskietce. Nazwa tego zbioru powinna mieć rozszerzenie ".OBJ". Następnie nazwę tę wprowadzamy do zbioru wskazówek dla linkera i wykonujemy przebieg konsolidacji. Do dołączonego w ten sposób podprogramu maszynowego można przekazać z programu w języku C argumenty, wykorzystując jeden bajt na szczycie stosu do przekazania informacji o liczbie argumentów.

Moduł uruchomieniowy DBC wykorzystuje pamięć RAM od adresu $3000 do $3FFF, a program użytkownika zaczyna się od adresu $4000 i ciągnie się aż do końca pamięci. Na procedury w kodzie maszynowym można przeznaczyć szóstą stronę pamięci i wolną pamięć pomiędzy końcem obszaru zajmowanego przez DOS a adresem $3000. Chociaż kompilator wymaga 48kB pamięci do pracy, to większość programów w DBC wymaga dużo mniej miejsca. Można na pewno napisać wiele programów, które zajmują tak niewiele pamięci jak 24kB RAM.

3. ZACZYNAMY PROGRAMOWAĆ

3.1. Podstawowe pojęcia: funkcje, komentarze, instrukcje.

Najlepszą metodą nauczenia się programowania w określonym języku jest ... pisanie w nim programów. Zatem napiszmy pierwszy, prosty program w języku C, który wyświetli na monitorze komunikat takiej treści: "Programowanie w języku C jest ciekawym zajęciem". Oto tekst źródłowy takiego programu:

Program 1

```
main()
{       /* początek funkcji main() */
    char c;
    printf("Programowanie w języku C jest ciekawym zajęciem");
    /* i tu właściwie kończy się ten program. Instrukcja  */
    /* char c; i poniższa instrukcja c=getchar() nie są   */
    /* potrzebne do wyprowadzenia komunikatu - patrz opis */
    /* w tekście.                                         */
    c=getchar();
}       /* koniec funkcji main() */
```

Kilka słów o sposobie kompilacji i konsolidacji tego programu. Tekst programu, dokładnie w takiej postaci, w jakiej jest on wydrukowany (oczywiście bez polskich znaków), wprowadzamy przy pomocy edytora tekstowego, np. ATARI WRITER do zbioru dyskowego o nazwie "PROG1.C". Przy okazji tworzymy zbiór wskazówek dla linkera, który będzie potrzebny w procesie konsolidacji naszego programu. Zbiór ten powinien mieć nazwę "PROG1.LNK" i zawierać dokładnie cztery niżej przedstawione wiersze:

```
PROG1
AIO
PRINTF
DBC.OBJ
```

ładujemy do pamięci komputera kompilator CC.COM, piszemy nazwę zbioru z programem źródłowym, tj. PROG1.C i naciskamy RETURN. Gdy kompilacja zakończy się komunikatem "no errors", ładujemy linker CLINK.COM. Gdy kompilacja zakończy się błędami, to weryfikujemy wprowadzony przez nas tekst programu z tekstem powyższym, poprawiamy błędy edytorem tekstowym i ponownie uruchamiamy kompilator. Linker uruchamiamy poleceniem L PROG1 i wciskamy RETURN. Linker utworzy nam wykonywalny kod programu, który zostanie zapisany do zbioru o nazwie PROG1.COM. Gdy załadujemy ten program do pamięci, zostanie on automatycznie wystartowany i wypisze na ekranie poniższy tekst:

Programowanie w języku C jest ciekawym zajęciem

Naciśnięcie dowolnego klawisza i RETURN spowoduje ponowne załadowanie DOS-a.

W postaci źródłowej programu w języku C można wyodrębnić trzy zasadnicze elementy:

1. nagłówek zawierający definicję stałych programu;
2. deklaracje zmiennych programu;
3. funkcje.

Program 1 nie zawiera nagłówka, natomiast zawiera deklarację jednej zmiennej o nazwie "c". Program ten składa się tylko z jednej funkcji o nazwie main(). Każdy program w języku C składa się przynajmniej z jednej funkcji, lecz może mieć ich również więcej. Jeżeli program zawiera tylko jedną funkcję, to musi ona mieć nazwę main(). Gdy program zawiera wiele funkcji, to jedna z nich musi mieć nazwę main(). Funkcja jest częścią programu wykonującą pewną operację, na przykład: wyprowadzenie informacji na ekran, obliczenie wartości wyrażenia, itp. Funkcja wykonuje operacje na przekazanych do niej wartościach zmiennych. Wartości te nazywane są w języku C argumentami funkcji. Listę argumentów umieszcza się w nawiasach okrągłych (..), tuż za nazwą funkcji. Jeżeli funkcja do wykonania operacji nie potrzebuje żadnych argumentów (funkcja bezargumentowa), to za jej nazwą umieszcza się tylko nawiasy, na przykład funkcja main() jest funkcją

bezargumentową. Nawiasy za nazwą funkcji muszą zawsze występować, gdyż umożliwiają one kompilatorowi DBC identyfikację, czy dana nazwa jest nazwą zmiennej (bez nawiasów), czy nazwą funkcji. Funkcja może przekazać z powrotem do funkcji, z której została wywołana, wynik wykonanych obliczeń czy operacji. Ten wynik nazywa się "wartością zwracaną przez funkcję". W języku C funkcja może mieć wiele argumentów, jednak zwracać może tylko jedną wartość. Funkcje zbudowane są z instrukcji języka C. Instrukcją w języku C jest wyrażenie zakończone średnikiem. Natomiast wyrażeniem jest każda konstrukcja zbudowana ze stałych, nazw zmiennych i stałych oraz operatorów. Zatem wyrażeniami w C są: 2+3, a+b, a>b, c, itp. Każda funkcja w języku C ma swoją unikalną nazwę, nadaną przez programistę. Jedynie nazwy funkcji zawartych w bibliotece kompilatora posiadają już nadane nazwy i programiście nie wolno ich zmieniać. Nazwa każdej funkcji musi spełniać poniższe warunki:

1. musi zaczynać się od litery (znak podkreślenia "_" jest traktowany w C jak litera);

2. musi zawierać tylko litery lub liczby (inne znaki nie są dopuszczalne w nazwie funkcji);

Poprawnymi nazwami funkcji są: wynik(), WYNIK(), a1(), a_1(). Nazwami niedopuszczalnymi są: Zła(), liczba-5(). Należy zwrócić uwagę, że nazwy wynik() i WYNIK() traktowane są przez kompilator jako nazwy różne.

Po nazwie funkcji występuje nawias otwierający "$(", a na jej końcu nawias zamykający "$)". Wszystko,co jest pomiędzy tymi nawiasami, jest traktowane przez kompilator jako jedna funkcja.

Program źródłowy w języku C może mieć dowolny układ, najbardziej odpowiadający indywidualnym upodobaniom każdego programisty. Przy jego redagowaniu należy jednak przestrzegać pewnych zasad:

- wszystkie słowa występujące w jednej linii programu muszą być oddzielone od siebie jedną lub kilkoma znakami spacji;

- instrukcje muszą być zakończone średnikiem;

- znak "#" (hash) stojący przed dyrektywą kompilatora (#define lub #include) musi być umieszczony zawsze na początku linii (musi być pierwszym znakiem w linii).

Tekst zawarty pomiędzy znakami /$....$/ jest w języku C komentarzem i jest ignorowany przez kompilator. Komentarz może być dowolnie długi. Powyższe zasady umożliwiają takie redagowanie programu źródłowego w języku C, że jego tekst może z powodzeniem stanowić dokumentację programu. Ma to istotne znaczenie w przypadku poprawiania napisanego programu po upływie pewnego czasu.

Instrukcje:

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

nie są konieczne do poprawnego działania programu 1. Jednakże program skompilowany kompilatorem DBC ma tę cechę, że w momencie, gdy wykonana zostanie ostatnia instrukcja programu następuje automatyczne ładowanie DOS-u. Zatem powyższe instrukcje wprowadzono do programu wyłącznie po to, aby móc zaobserwować efekt działania programu na ekranie monitora. Więcej o tych instrukcjach powiemy w dalszej części tego rozdziału.

Rozważmy kolejny program:

Program 2

```
main()
 ${     /$ początek funkcji main() $/
 char c;
 printf("Nazywam się Jan Kowalski.");
 printf(A Ty jak się nazywasz?");
 c=getchar();
$)      /$ koniec funkcji main() $/
```

Po jego uruchomieniu zobaczymy na ekranie poniższy komunikat:

Nazywam się Jan Kowalski.A Ty jak się nazywasz?

Załóżmy, że chcemy, aby drugie zdanie komunikatu zaczynało się od nowej linii. W tym celu zmieniamy linię z programu 2

printf("Nazywam się Jan Kowalski.");

na linię:

printf("Nazywam się Jan Kowalski.\n");

Po takiej modyfikacji uzyskamy na ekranie tekst wyświetlony w dwóch wierszach:

```
Nazywam się Jan Kowalski.
A Ty jak się nazywasz?
```

Wstawiony na końcu linii symbol składający się z dwóch znaków "\n" (backslash i mała litera n) jest w języku C znakiem nowej linii. Pomiędzy znakami "\" i "n" nie może być spacji. Znak nowej linii może być użyty w dowolnym miejscu wyprowadzanego komunikatu i spowoduje, że tekst występujący po nim zostanie wyświetlony na ekranie od początku nowej linii. A więc identyczny efekt uzyskalibyśmy, zmieniając zamiast pierwszej - drugą linię printf(...), nadając jej postać:

printf("\nA Ty jak się nazywasz?");

Oprócz znaku nowej linii kompilator DBC dopuszcza stosowanie innych znaków mających specjalne znaczenie:

\f - czyść ekran;

\g - włącza sygnał dźwiękowy;

\h - kasuje znak z lewej strony aktualnej pozycji kursora i przesuwa kursor o jedną pozycję w lewo;

\r - kasuje całą linię;

\\ - wyprowadza znak backslash;

\' - wyprowadza apostrof;

\" - wyprowadza cudzysłów;

\t - wyprowadza znak tabulacji;

\### - wyprowadza znak, którego kod ATASCII jest trzycyfrową liczbą oktalną o postaci ###. Np.: litera "A" ma kod ATASCII równy 65 a odpowiada mu liczba oktalna 101. Zatem użycie w komunikacie sekwencji \101 spowoduje wyprowadzenie litery A.

W łańcuchach znaków wyprowadzanych przy pomocy funkcji wyjścia sformatowanego printf() można umieszczać dowolne znaki (również w inwersie video) poza jednym wyjątkiem. Znak "CTRL-" (serduszko) wykorzystywany jest przez kompilator do oznaczenia końca łańcucha znaków i z tego powodu nie powinien być w nim używany. Znak "\" (backslash) również ma specjalne znaczenie, wyżej przedstawione, o którym nie należy zapominać.

3.2. Stałe i zmienne. Typy i deklaracje.

Chcemy dodać do siebie dwie liczby i ich sumę wyprowadzić na monitor. Oto program realizujący to zadanie:

Program 3

```
main()
 ${          /$ początek funkcji main() $/
 char c;
 int suma;   /$ deklaracja zmiennej suma $/
 suma=15+34;
 printf("suma jest równa: %d\n",suma);
 c=getchar();
 $)          /$ koniec funkcji main()    $/
```

W programie 3 "suma"jest nazwą zmiennej. Zmienne w języku C, podobnie jak w innych językach programowania są obiektami, na których program wykonuje operacje. Każdą zmienną kompilator DBC identyfikuje poprzez nazwę. Nazwy zmiennych w języku C podlegają identycznym ograniczeniom jak nazwy funkcji (patrz 3.1.)

Kompilator DBC musi być "poinformowany", że programista

będzie używał w programie zmiennych. Do przekazywania mu tych informacji służą deklaracje zmiennych. Innymi słowy każda zmienna w programie w języku C musi być zadeklarowana przed jej użyciem. Instrukcja "int suma;" jest deklaracją. Wszystkie deklaracje zmiennych muszą wystąpić na początku funkcji, tuż za nawiasem otwierającym "$(" (są wyjątki od tej zasady). W jednej instrukcji będącej deklaracją można wymienić kilka nazw zmiennych tego samego typu np. instrukcja:

```
int suma,sumai,a,b,c;
```

jest poprawną deklaracją pięciu zmiennych typu całkowitoliczbowego o nazwach : suma, sumai, a, b, c.

DBC dopuszcza deklarację trzech typów zmiennych:

char - oznacza zmienną typu znakowego;

int - oznacza zmienną numeryczną całkowitoliczbową;

zmienna wskaźnikowa (typ pointer) o których powiemy nieco później.

Zmienna suma została zadeklarowana jako zmienna typu całkowitoliczbowego, co oznacza, że można w niej przechowywać liczby całkowite, dodatnie lub ujemne należące do przedziału [-32768, +32767]. Zmienna typu int w implementacji DBC zajmuje dwa bajty pamięci, przy czym liczba zapisana jest na piętnastu bitach, a jeden bit wykorzystywany jest jako bit znaku. Zmienna c została zadeklarowana w programie 3 jako zmienna typu znakowego. Ten typ zmiennych w implementacji DBC zajmuje jeden bajt pamięci i jest traktowany jako liczba dodatnia bez znaku z zakresu [0, 255].

Wszystkim zadeklarowanym zmiennym kompilator DBC przypisuje początkową wartość równą zero. Do zmiany wartości określonej zmiennej służy instrukcja przypisania, zbudowana w oparciu o operator przypisania "=". W instrukcji tej zmienna, której chcemy nadać wartość, występuje po lewej stronie znaku równości, natomiast po prawej stronie znaku umieszcza się wyrażenie, którego wartość zostanie przypisana zmiennej. Zatem instrukcja: suma=15+24; w języku C należy czytać: "przypisz zmiennej suma wartość wyrażenia 15+25". Operatorem przypisania jest znak równości. W języku C istnieje możliwość tworzenia również innych operatorów przypisania. Umożliwia to konstrukcja "op=", gdzie "op" jest jednym z następujących operatorów dwuargumentowych: +, -, *, /, %, <<, >>, &, ^, |. Jak kompilator DBC interpretuje utworzone w ten sposób operatory i w jakich sytuacjach wolno ich stosować? Wyjaśnimy to na przykładzie: jeśli mamy instrukcję przypisania typu a=a+b; to możemy ją zapisać w taki sposób: a+=b. Odnosi się to również do instrukcji a=a-b; a=a/b; a=a*b; itd. Wszystkie tego typu instrukcje mogą być zastąpione odpowiednimi instrukcjami: a-=b; a/=b; a*=b itd, przy czym b nie musi być tylko zmienną, lecz może być również wyrażeniem np.: instrukcja a=a+(c*2) jest równoważna instrukcji a+=c*2.

Program 3 można zapisać również tak:

Program 4

```
#define     LICZBA1     15
#define     LICZBA2     24
main()$(    /* początek main()    */
 int suma;
 char c;
 suma=LICZBA1+LICZBA2;
 printf("Suma jest równa: %d\n",suma);
 c=getchar();
$)          /* koniec main()  */
```

Efekt działania tego programu będzie identyczny jak programu 3. Wprowadzone zmiany dotyczą sposobu definiowania stałych w programie. W programie 3 stałe zostały zdefiniowane wprost, przez umieszczenie ich wartości w instrukcji programu. Jednak często dobrze jest używać stałych w programie poprzez nazwę, jak to ma miejsce w programie 4. #define jest dyrektywą kompilatora DBC, która informuje go, że gdy analizując program źródłowy, spotka nazwę wymienioną w tej dyrektywie ,to ma ją zastąpić wartością przypisaną tej nazwie. Nie dotyczy to oczywiście nazw występujących w łańcuchach znakowych. Definiowanie stałych za pomocą dyrektywy #define gwarantują, że w przypadku gdy zajdzie potrzeba zmiany wartości stałych, nie trzeba będzie zmieniać instrukcji programu (czasami wielu), a jedynie dyrektywę #define.

Dyrektywą #define można definiować następujące typy stałych:

1. dziesiętne np.: 173, -45;
2. oktalne np.: 017, 045;
3. szesnastkowe np.: 0xd400, 0xff, 0xC000;
4. znakowe np.: 'a', 'b', 'l';
5. łańcuchowe np.: "okno", "ala";

A oto poprawne przykłady definiowania różnych typów stałych przy wykorzystaniu dyrektywy #define:

```
#define     11      25              #define liczba  -1348
#define     12      0101            #define adres 0xe465
#define     znak    'T'             #define kom "cześć"
```

Przyjęło się, że nazwy stałych pisze się w języku C dużymi literami, jednak nie jest to konieczne. Zasada ta ma tę zaletę, że łatwo jest odróżnić w programie źródłowym stałe od zmiennych.

Zdefiniowane dyrektywą #define nazwy nie mogą być używane w programie jako zmienne. Nie można wykonywać na nich operacji przypisania i innych. Instrukcja LICZBA1=30; gdyby została umieszczona w programie 4 byłaby traktowana przez kompilator DBC jako błędna. Dyrektywy #define nie wolno kończyć średnikiem.

Oprócz dyrektywy #define kompilator DBC dopuszcza stosowanie jeszcze jednej dyrektywy, a mianowicie #include. Informuje ona kompilator, że ma on odczytać z dyskietki zbiór o nazwie podanej w dyrektywie i włączyć jego zawartość do kompilowanego programu. Włączony zbiór zawiera z reguły tekst źródłowy jednej lub kilku funkcji programu. Można jej również użyć do wczytania zbioru zawierającego definicję wszystkich stałych programu (zbiór taki zawiera tylko dyrektywy #define).

3.3. Decyzje programowe.

Bardzo często program musi podjąć określone działanie w zależności od zaistniałych warunków. Sytuację taką ilustruje poniższy problem: należy wyświetlić na monitorze pytanie, na które użytkownik może odpowiedzieć "tak" lub "nie". Jeżeli odpowiedź jest "tak", należy wyprowadzić określony komunikat; gdy odpowiedź jest "nie" - inny komunikat. Program działający w taki właśnie sposób wygląda następująco:

Program 5

```
main()$(                /* początek main */
char c,odp;
printf("Chcesz programować w języku C? (T/N) \n");
odp=getchar();          /* wczytaj odpowiedź */
c=getchar();            /* wczytaj return    */
  if(odp=='T')$(        /* początek "if"     */
  printf("Czytaj dalej ten rozdział\n");
$)                      /* koniec "if"       */
  if(odp=='N')$(        /* początek  "if"    */
  printf("To nie męcz się dłużej\n");
$)                      /* koniec "if"       */
c=getchar();
$)                      /* koniec main()     */
```

Program 5 zbudowany jest z jednej funkcji main(), z której wywoływane są dwie funkcje z biblioteki DBC, poznana wcześniej funkcja printf() i funkcja getchar() z biblioteki AIO. Funkcja getchar() działa podobnie jak instrukcja INPUT z BASIC-a, tzn. praca komputera zostaje zawieszona aż do momentu, gdy użytkownik naciśnie klawisz RETURN. Kod klawisza naciśniętego przed naciśnięciem RETURN zwracany jest do funkcji wywołującej funkcję getchar(). Stąd instrukcja przypisania:

```
odp=getchar();
```

powoduje, że zmiennej "odp" przypisana zostaje wartość zwrócona przez funkcję getchar(). Program 5 po uruchomieniu wyświetla pytanie: "Chcesz programować w języku C? (T/N)" i czeka aż użytkownik naciśnie jeden z klawiszy T lub N. Gdy użytkownik naciśnie T program wyświetli tekst "Czytaj dalej ten rozdział" - gdy N - tekst "To nie męcz się dłużej". Takie rozstrzygnięcie przez program, jaka akcja ma być podjęta, nazywa się decyzją programową.

3.3.1. Instrukcja if.

Decyzja programowa podejmowana jest w oparciu o wynik badania warunku lub inaczej, zdarzenia programowego. Do badania warunków (zdarzeń programowych) służy w języku C, między innymi instrukcja if, będąca jedną ze słów kluczowych tego języka. W tym miejscu wypada uczynić również zastrzeżenie, że nazwy zmiennych, stałych i funkcji muszą być różne od słów kluczowych języka C. Wykaz wszystkich słów kluczowych DBC wymieniony został we wstępie.

Składnia instrukcji if jest następująca:

```
if(warunek)
${       /* nawias otwierający          */
   instrukcja1;
   instrukcja2;
   .......
$}       /* nawias zamykający          */
instrukcja;   /* ta instrukcja wykonywana jest wówczas
               * gdy warunek instrukcji if(..) nie
               * jest spełniony                      */
```

Po słowie kluczowym if musi być zdefiniowany warunek ujęty w nawiasy (...). Warunek definiowany jest przy pomocy wyrażenia warunkowego, które może przyjmować dwie wartości: TRUE i FALSE. Wyrażeniem warunkowym jest:

```
odp=='T'
```

a odczytuje się je: "czy wartość zmiennej odp jest równa 'T'". Gdy wyrażenie warunkowe przyjmuje wartość TRUE, to wykonywana jest sekwencja instrukcji ujętych w nawiasy ${..$}. Gdy przyjmie wartość FALSE, wykonywana jest pierwsza instrukcja występująca po nawiasie zamykającym. Nawiasy ${ i $} mogą być pominięte jedynie wówczas, gdy przy spełnionym warunku ma być wykonana tylko jedna instrukcja.

W języku C wartości logicznej TRUE odpowiada wartość różna od zera, natomiast wartości logicznej FALSE odpowiada zero. Dzięki tej własności wyrażenia warunkowe w języku C mogą być budowane z instrukcji przypisania, wyrażeń arytmetycznych czy wartości zwracanych przez funkcje. Oczywiście można je również konstruować w klasyczny sposób, jak w języku BASIC, wykorzystując do tego operatory relacji, którymi są:

== oznacza "jest równe";

!= oznacza "jest różne";

> oznacza "jest większe";

< oznacza "jest mniejsze";

>= oznacza "jest większe lub równe";

<= oznacza "jest mniejsze lub równe".

A oto kilka przykładów wyrażeń warunkowych zbudowanych przy wykorzystaniu operatorów relacji:

10==7+3 - "czy 10 jest równe 7+3";

ala1>=ala2 - "czy wartość zmiennej ala1 jest większa lub równa wartości zmiennej ala2;

Powyższe wyrażenia warunkowe nazywane są prostymi wyrażeniami. W języku C można konstruować również złożone wyrażenia warunkowe przy pomocy wyrażeń prostych i operatorów logicznych, którymi są:

|| - oznacza logiczną alternatywę (OR);

&& - oznacza logiczną koniunkcję (AND);

! (wykrzyknik) - oznacza logiczną negację (NOT).

Przy konstruowaniu złożonych wyrażeń warunkowych można używać nawiasów (..) do wskazywania kolejności obliczania wartości poszczególnych wyrażeń prostych. Przykładami złożonych wyrażeń warunkowych są:

```
a>='0' && a<='9'    "czy a jest cyfrą"
a!='T' && a!='t'    "czy a nie jest literą t"
```

Również poprawnymi w języku C są konstrukcje warunkowe:

```
if(5);      if(c=a);      if(czydalej());
```

pomimo że wyrażenia warunkowe w nich zawarte nie są zbudowane z operatorów relacji. Pierwsze z nich przyjmuje zawsze wartość logiczną TRUE (ponieważ 5 jest różne od zera), drugie przyjmie wartość logiczną FALSE jedynie wówczas, gdy zmiennej c zostanie przypisana wartość zmiennej a równa zero, a trzecie przyjmie wartość logiczną FALSE jedynie wówczas, gdy funkcja czydalej() zwróci do miejsca wywołania zero.

Do budowy wyrażeń służą również w języku C bitowe operatory logiczne, pozwalające na manipulację pojedyńczymi bitami ich argumentów. Do operatorów tych należą:

& - bitowa koniunkcja;

| - bitowa alternatywa;

^ - bitowa różnica symetryczna;

<< - przesunięcie w lewo;

>> - przesunięcie w prawo;

~ - uzupełnienie jedynkowe.

Działanie tych operatorów ilustrują poniższe tabelki, przy założeniu, że zmienne x, y, z są typu char oraz x='A' i y='C'.

bit	7	6	5	4	3	2	1	0
arg. x	0	1	0	0	0	0	0	1
arg. y	0	1	0	0	0	0	1	1
z=x&y	0	1	0	0	0	0	0	1

bit	7	6	5	4	3	2	1	0
arg. x	0	1	0	0	0	0	0	1
arg. y	0	1	0	0	0	0	1	1
z=x|y	0	1	0	0	0	0	1	1

bit	7	6	5	4	3	2	1	0
arg. x	0	1	0	0	0	0	0	1
arg. y	0	1	0	0	0	0	1	1
z=x^y	0	0	0	0	0	0	1	0

bit	7	6	5	4	3	2	1	0
arg. x	0	1	0	0	0	0	0	1
z=~x	1	0	1	1	1	1	1	0

Operatory << i >> służą do przesuwania bitów argumentu stojącego po lewej stronie operatora o liczbę pozycji stojącą po prawej stronie. Na przykład: x<<2 przesuwa zawartość zmiennej x o dwa bity w lewo. Zwolnione w wyniku przesunięcia pozycje bitowe zmiennej x wypełniane są zerami.

Tworzone za pomocą wyżej przedstawionych operatorów wyrażenia warunkowe jak: if(a&b), if(a|b), itp są również akceptowane przez DBC.

Gdy dokładniej przeanalizujemy działanie programu 5, to stwierdzimy, że posiada on pewne wady. Gdy użytkownik wciśnie małe "t" lub małe "n", program nie podejmie żadnego działania. Znając zasady konstruowania wyrażeń warunkowych, możemy go zatem poprawić, aby wyeliminować tę wadę.

Program 6

```
#define TD      'T'
#define TM      't'
#define ND      'N'
#define NM      'n'
main()${
     char c,odp;
     printf("Chcesz programować w języku C? (T/N)\n");
     odp=getchar();
     c=getchar();
     if(odp==TD || odp==TM)
          printf("Czytaj dalej ten rozdział\n");
     if(odp==ND || odp==NM)
          printf("to nie męcz się dłużej\n");
     c=getchar();
$}        /* koniec main()    */
```

Zasadnicze zmiany wprowadzone w tym programie w stosunku do programu 5 są następujące:

- stałe znakowe 'T' i 'N' zostały zdefiniowane w dyrektywie #define;

- zdefiniowane zostały dwie dodatkowe stałe znakowe 't' i 'n'

po to, aby program działał bez względu na to, czy w odpowiedzi naciśnięto dużą czy małą literę;

- usunięto nawiasy $(i $) ograniczające zakres instrukcji if. Było to możliwe dlatego, że zarówno przy spełnieniu pierwszego jak i drugiego warunku wykonywana jest tylko jedna instrukcja;
- wprowadzono złożone wyrażenia warunkowe w miejsce prostych.

Przedstawimy teraz dwie pożyteczne funkcje z biblioteki kompilatora DBC, dzięki którym złożone wyrażenia warunkowe z programu 6 będziemy mogli ponownie zastąpić wyrażeniami prostymi. Funkcjami tymi są: tolower() i toupper(). Funkcja tolower() zwraca kod ATASCII małej litery, gdy jej argumentem jest kod dużej litery. Gdy jej argumentem jest kod małej litery, to jest on zwracany bez zmian. Natomiast funkcja toupper() zwraca kod ATASCII dużej litery, gdy jej argumentem jest kod małej litery. Kod dużej litery jest zwracany przez tę funkcję bez zmian. Wykorzystując jedną z tych dwóch funkcji program 6, możemy zmodyfikować nadając mu postać:

Program 7

```
#define TD      'T'
#define ND      'N'
main()$(
     char c,odp;
     printf("Chcesz programować w języku C? (T/N)\n");
     odp=toupper(getchar());
     c=getchar();/$ przeskocz RETURN $/
     if(odp==TD)
          printf("Czytaj dalej ten rozdział\n");
     if(odp==ND)
          prinft("to nie męcz się dłużej\n");
     c=getchar();
$)        /$ koniec main()   $/
```

W programie tym nie ma sprawdzania, czy odpowiedź na wyświetlone pytanie udzielona była przy użyciu dużej czy małej litery. Stało się to możliwe dzięki instrukcji przypisania:

```
odp=toupper(getchar());
```

po wykonaniu której zmienna "odp" zawsze zawierać będzie kod ATASCII odpowiadający dużej literze alfabetu, o ile naciśnięty został klawisz odpowiadający literze (A - Z). W sytuacji, gdy naciśnięty zostanie klawisz nie będący literą, np. cyfra, SHIFT, RETURN itp. zmienna "odp" zawierać będzie niezmieniony kod tego klawisza.

Pozostańmy nadal przy programie 7. Jego dokładna analiza prowadzi do wniosku, że nie działa on poprawnie w każdej sytuacji. Co ten program "zrobi", gdy na wyświetlone pytanie użytkownik naciśnie, świadomie lub pomyłkowo, klawisz inny niż "T" lub "N". Program nie wyświetli w takiej sytuacji żadnego komunikatu i zakończy się. Poprawmy go zatem tak, aby uwzględniał on również ten przypadek. Oto jego nowa wersja:

Program 8

```
#define TD      'T'
#define ND      'N'
main()$(
     char c,odp;
     printf("Chcesz programować w języku C? (T/N)\n");
     odp=toupper(getchar());
     c=getchar();/$ przeskocz RETURN $/
     if(odp==TD)
          printf("Czytaj dalej ten rozdział\n");
     else if(odp==ND)
          prinft("to nie męcz się dłużej\n");
     else
     $(
     printf("\nUdzieliłeś niewłaściwej odpowiedzi");
     printf("\nmiałeś nacisnąć T lub N");
     printf("\na nacisnąłeś '%c'",odp);
     $)
     c=getchar();
$)        /$ koniec main()   $/
```

Jak widać, aby poprawić nasz program, musieliśmy użyć nowej instrukcji else.

3.3.2. Konstrukcja if-else

Konstrukcje programowe typu if-lse mają zastosowanie w sytuacji, gdy program musi wykonać pewne działanie przy spełnionym określonym warunku a inne działanie, gdy warunek ten jest niespełniony. Działanie takich konstrukcji programowych można zilustrować poniższym schematem:

```
if(wyrażenie)        /$ gdy to wyrażenie przyjmuje
     $(              $ wartość logiczną TRUE
     działanie1;     $ wykonywane jest to działanie
     $)
else                 $ w przeciwnym wypadku
     $(
     działanie2;     $ program podejmuje tę akcję
     $)              $/
```

Ze schematu tego wynika, że konstrukcje if-else służą do wyboru jednej z dwóch możliwych ścieżek przetwarzania. Jednak w praktyce programowania często spotyka się bardziej skomplikowane sytuacje, gdy program musi wybrać jedną z kilku możliwych akcji. W języku C istnieje konstrukcja programowa umożliwiająca programiście wygodne kodowanie takich sytuacji. Konstrukcja ta zbudowana jest również ze słów kluczowych if i else, jednak dla odróżnienia jej od wyżej przedstawionej, określa się ją niekiedy nazwą else-if. Konstrukcja else-if na postać poniższą:

```
if(wyrażenie1)       /$ gdy to wyrażenie ma wartość
     $(              $ logiczną TRUE
     akcja1;         $ program wykonuje tę akcję
     $)              $
else if(wyrażenie2)  $ gdy to wyrażenie ma wartość
     $(              $ logiczną TRUE
     akcja2;         $ program wykonuje tę akcję
     $)              $
.........            $ itd
else                 $ gdy wszystkie w/w wyrażenia
     $(              $ mają wartość logiczną FALSE
     akcja;          $ program wykonuje tę akcję
     $)              $/
```

Nawiasy ograniczające $(..$) mogą być pominięte, gdy określona akcja składa się tylko z jednej instrukcji.

Omawiając konstrukcje sterujące programem (do takich należą if-else i else-if), należy dodać, że implementacja DBC nie posiada rozkazu skoku (goto), chociaż standardowy język C rozkaz taki zawiera. Ma to swoje dobre i złe strony. Rozkazy skoku są nagminnie nadużywane we wszystkich językach programowania, w których są dostępne. DBC jest wyposażony w podstawowe konstrukcje sterujące, umożliwiające wybór właściwej drogi przetwarzania. W praktyce instrukcja skoku nigdy nie jest konieczna i zawsze można się bez niej obejść. Zatem jej brak w implemmtacji DBC zmusza programistę do stosowania dobrego stylu programowania - programowania strukturalnego. Wadą spowodowaną jej brakiem jest konieczność powtarzania sprawdzeń warunków i większa liczba zmiennych programowych.

Wracając jednak do naszego programu musimy stwierdzić, że jego działanie nie dostarcza nam jeszcze pełnej satysfakcji. Program 8 poinformuje użytkownika o naciśnięciu niewłaściwego klawisza i co dalej? Nic, zakończy działanie. Jednak dobrze byłoby, gdyby nasz program ponowił w takiej sytuacji pytanie.

3.3.3 Konstrukcja do-while.

Sytuacja, w której program musi powtórzyć przyjęcie określonej informacji od użytkownika, bardzo często występuje w praktyce programowania. Na przykład DOS: wyświetlając swoje menu oczekuje na polecenie wybierające jedną z możliwych opcji. Gdy użytkownik naciśnie literę nie odpowiadającą żadnej opcji, zostanie poinformowany o niewłaściwym wyborze i DOS ponownie będzie oczekiwał na komendę. Ta procedura będzie wykonywana tak długo, aż użytkownik wybierze poprawną (możliwą do wykonania) opcję.

Język C dostarcza programiście narzędzi umożliwiających powtarzanie przez program określonej akcji dowolną ilość razy. Narzędziem tym jest konstrukcja do - while. Konstrukcja ta zbudowana jest ze słów kluczowych do i while i jej postać jest następująca:

```
do
$(
działanie;
$)
while(wyrażenie);
```

Po słowie kluczowym do następuje ciąg instrukcji ujętych w

nawiasy @(..@), który jest wykonywany tak długo, jak długo wyrażenie ujęte w nawiasy (..), występujące po słowie kluczowym while przyjmuje wartość różną od zera. W momencie, gdy wyrażenie to przyjmie wartość zero, sterowanie w programie przenoszone jest do pierwszej instrukcji występującej po słowie while i związanym z tym słowem wyrażeniem. Zatem aby program 8 ponawiał pytanie w przypadku naciśnięcia klawisz innego niż "t" lub "n", powinien mieć poniższą postać:

Program 9

```
#define TAK     'T'
#define NIE     'N'
main()@(
     char c,odp;
     do
@(/* początek do  */
printf("Chcesz programować w języku C? (T/N)\n");
odp=toupper(getchar());
c=getchar();/* przeskocz RETURN */
if(odp!=TAK && odp!=NIE)@(
 printf("\nUdzieliłeś niewłaściwej odpowiedzi");
 printf("\nmasz nacisnąć T lub N");
 printf("\na nacisnąłeś '%c'",odp);
@)/* koniec if*/
@)/* koniec do  */
while(odp!=TAK && odp!=NIE);
/* od tego miejsca rozpocznij wykonywanie programu,
 * gdy zostanie naciśnięty jeden z klawiszy "t"
 * lub "n" */
if(odp==TAK)
     printf("Czytaj dalej ten rozdział\n");
else
     printf("to nie męcz się dłużej\n");
c=getchar();
@)        /* koniec main()   */
```

Program 9 po wyświetleniu pytania czeka na odpowiedź. Gdy naciśnięty zostanie klawisz inny niż "t" lub "n", program wyświetli komunikat o udzieleniu niewłaściwej odpowiedzi i ponownie wyświetli pytanie. Akcję tę program będzie kontynuował tak długo, aż zostanie naciśnięty albo klawisz "t" albo "n". Wówczas nastąpi wyjście z pętli do - while i wyprowadzony zostanie jeden z dwóch, przewidzianych w takiej sytuacji komunikatów.

3.3.4 Instrukcja while.

Przedstawiona w poprzednim rozdziale pętla programowa może być również skonstruowana inaczej - w sposób bardziej elegancki. Szczegółowa analiza pętli do - while z programu 9 prowadzi do następujących wniosków:

- wyświetlane jest pytanie: "Czy chcesz...?.";
- wczytywany jest znak z klawiatury;
- badane jest (instrukcja if), czy wczytany znak nie jest ani literą T ani N;
- jeżeli nie jest, to wyświetlany jest komunikat o udzieleniu niewłaściwej odpwiedzi;
- ponownie badane jest, czy wczytany znak jest inny niż T i N (tym razem badanie tego warunku realizowane jest w instrukcji while);
- jeżeli jest inny, to pętla wykonywana jest od początku.

Nieracjonalnym w tej pętli jest dwukrotne badanie tego samego warunku. Jest to cena, którą trzeba płacić za brak instrukcji skoku. Ale czy naprawdę trzeba? Nie zawsze; program 10 zawiera pętlę realizującą to samo zadanie co program 9, jednak wyeliminowano z niego dwukrotne badanie tego samego warunku. Do zbudowania tej pętli zastosowana została instrukcja while, będąca również słowem kluczowym języka C. Postać tej instrukcji jest następująca:

```
while (wyrażenie logiczne)
@(
     instrukcja1
     instrukcja2
     ...
@)
```

Słowo kluczowe while otwiera pętlę. Tuż za nim musi wystąpić wyrażenie logiczne ujęte w nawiasy (..). Bezpośrednio za wyrażeniem logicznym występuje ciąg instrukcji ujętych w nawiasy @(..@). Jeśli wystąpi tylko jedna instrukcja nawiasy można pominąć. Instrukcja while działa w ten sposób, że na początku obliczana jest wartość wyrażenia logicznego i jeśli

Program 10

```
#define   TAK   'T'
#define   NIE   'N'
main()
@(
 char odp,ret;
 printf("Czy chcesz programować w języku C?\n");
 printf("odpowiedz T lub N\n");
 printf("--> ");
 odp=toupper(getchar());
 ret=getchar();  /* pomin RETURN */
 while(odp!=TAK && odp!=NIE)
     @(   /* początek petli while */
     printf("\nUdzieliłeś złej odpowiedzi %c",odp);
     printf("\nnacisnij T lub N");
     printf("--> ");
     odp=toupper(getchar());
     ret=getchar();
     @)   /* koniec petli while */
 if(odp==TAK)
   printf("\nCzytaj dalej ten rozdział");
 else
   printf("\nTo nie męcz się dłużej");
 odp=getchar()  /* zawies program przed */
                /* załadowaniem DOS-a */
@)              /* koniec main() */
```

jest ona różna od zera (TRUE), wykonywane są instrukcje zawarte w pętli (pomiędzy nawiasami @(..@)). Proces ten powtarzany jest tak długo, aż wyrażenie logiczne nie przyjmie wartości zerowej (FALSE). Wówczas żadna z instrukcji zawartych w pętli nie zostanie wykonana, a sterowanie zostanie przeniesione do instrukcji występującej po nawiasie zamykającym pętlę.

Warto zapamiętać podstawową różnicę w realizacji pętli do - while i while. W pierwszym przypadku instrukcje składające się na pętlę wykonywane są przed sprawdzeniem warunku. Zatem zawsze pętla do - while zostanie wykonana przynajmniej jeden raz. Natomiast w drugim przypadku, najpierw następuje sprawdzenie warunku i w zależności od jego wyniku pętla może zostać wykonana lub nie. Zatem w szczególnej sytuacji pętla while może być w ogóle nie wykonana.

Uważnego czytelnika intryguje zapewne średnik stojący tuż za nawiasem zamykającym wyrażenie warunkowe pętli while; zadaje on sobie pytanie: dlaczego wyrażenie warunkowe w instrukcji if nie jest zakończone średnikiem, a w instrukcji while jest? Skąd się bierze taka niekonsekwencja? Otóż, nie jest to brak konsekwencji, lecz wprost przeciwnie, średnik jest dowodem wnikliwego przemyślenia przez twórców języka C wszystkich stosowanych w nim konstrukcji programowych. Instrukcja while występująca w pętli do - while i w pętli while jest tą samą konstrukcją języka C. Zatem kompilator, interpretując kod źródłowy programu oczekuje wystąpienia po słowie kluczowym while ciągu instrukcji do wykonania. Gdyby ich nie znalazł sygnalizowałby błąd. Aby go "oszukać", umieszcza się po słowie kluczowym while w pętli do - while instrukcję minimalną dla całego programu, tzw. instrukcję pustą. Taką właśnie instrukcją jest sam średnik.

Przedstawione instrukcje if, else, do, while nie wyczerpują wszystkich możliwości sterowania programem w języku C. O ile instrukcja if jest wygodna przy wyborze jednej z dwóch możliwych ścieżek przetwarzania, to jej zastosowanie przy wyborze jednej z wielu dróg staje się już mniej wygodne, a ponadto ogranicza czytelność programu źródłowego. Język C posiada również dużo wydajniejsze instrukcje ułatwiające sterowanie programem i organizowanie pętli programowych - instrukcje switch , for, które zostaną przedstawione w następnych rozdziałach.

4. OPERACJE NA LICZBACH

Przykładowe programy przedstawione w poprzednim rozdziale zawierały proste operacje artymetyczne. Język C umożliwia realizowanie również skomplikowanych operacji arytmetycznych, podobnie jak inne języki wysokopoziomowe (Pascal, Basic). Wystarczy zadeklarować nazwy stałych i zmiennych występujących w wyrażeniu arytmetycznym i skonstruować to wyrażenie, wykorzystując niżej przedstawione operatory arytmetyczne:

+ - dodawanie;

(-) - odejmowanie;

* - mnożenie;

/ - dzielenie.

Wartość wyrażenia arytmetycznego może być przypisana do pewnej zmiennej programowej lub wyprowadzona bezpośrednio, np. na ekran. Ilustruje to poniższy program:

```
main()
{
 int a,b,c;
 a=20;
 b=25;
 c=a+b;
 printf("a+b = %d ",c);
}
```

Po uruchomieniu tego programu na ekranie zostanie wyświetlona linia

```
a+b = 45
```

Można oczywiście obliczać wartości bardziej skomplikowanych wyrażeń, np.:

```
b=20;
c=25;
d=-10;
a=b+c*d;
```

W przykładzie tym zmiennej d została przypisana wartość minus 10 (-10). Przypisanie to wykorzystuje jednoargumentowy operator (-), tzw. minus unarny. Operator ten działa według zasady: "pobierz wartość argumentu i zmień jej znak na przeciwny". W przykładzie tym widać również, że w jednej linii (instrukcji) może być użytych wiele operatorów arytmetycznych. Następny przykład ilustruje nowy problem:

```
b=20;
c=25;
d=3;
a=b+c*d;
```

Jak w języku C zostanie wykonana taka operacja? Czy wartość b zostanie dodana do c i następnie suma pomnożona przez d? A może inaczej: najpierw c zostanie pomnożone przez d i iloczyn zostanie dodany do wartości b.

W języku C istnieją ściśle określone reguły, według których obliczana jest wartość wyrażeń. Reguły te definiują ważność każdego operatora, a co za tym idzie, kolejność wykonywania operacji.

Największą wagę mają operatory mnożenia i dzielenia i te działania wykonywane są zawsze jako pierwsze (licząc od lewej do prawej strony wyrażenia). Następne w kolejności są operatory dodawania i odejmowania. Ponieważ C umożliwia stosowanie operatorów arytmetycznych i logicznych w jednym wyrażeniu, stąd następnymi w kolejności są operatory logiczne porównania (==,>,< itd.), koniunkcji (&&), alternatywy (||). Na samym końcu w tej hierarchii ważności znajduje się operator przypisania (=).

Wracając do powyższego przykładu, możemy teraz stwierdzić, że najpierw wartość c zostanie pomnożona przez d i iloczyn zostanie dodany do wartości b. Obliczony w ten sposób wynik (-95) zostanie przypisany zmiennej a. Rozważmy wyrażenie:

```
if (a==3+8 && b>20)
```

Opierając się na wyżej przedstawionych zasadach przeanalizujemy, jak obliczona zostanie jego wartość:

- w pierwszej kolejności wykonana zostanie operacja dodawania 3+8 i wynik 13 zostanie gdziekolwiek zapamiętany;

- następnie zostaną sprawdzone warunki: czy wartość zmiennej a jest równa 13, a wartość zmiennej b jest większa od 20. Załóżmy, że a=13 a b=28. Ponieważ przy tym założeniu obydwa wyrażenia logiczne przyjmą wartość logiczną TRUE, zatem i całe wyrażenie przyjmie taką wartość.

Należy zapamiętać, że w języku C kolejność wykonywania operacji, zarówno arytmetycznych jak i logicznych jest następująca:

działanie	wykonywane
* i /	przed
+ i -	przed
==, !=, >, <, >=, <=	przed
&& i ||	przed
= (przypisanie)	

Kolejność ta może być zmieniona przez zastosowanie nawiasów. Wracając do przedstawionego przykładu, jeśli wyrażenie arytmetyczne napiszemy stosując nawiasy:

```
a=(b+c)*d
```

to zawartość zmiennej b zostanie dodana do c i suma pomnożona przez wartość zmiennej d, a zatem inaczej niż bez użycia nawiasów. Nawiasy mogą być stosowane w dowolny sposób; ważne, aby ich liczba była parzysta.

Przedstawimy teraz program realizujący funkcję prostego kalkulatora. Działał on będzie w następujący sposób:

-zapytaj użytkownika o rodzaj operacji, która ma być wykonana;

-zapytaj go o dwa argumenty, na których należy wykonać wybraną operację;

-wykonaj operację;

-wyświetl wynik operacji;

-powtórz wszystko od początku, gdy użytkownik wyrazi takie życzenie. Na początek zajmiemy się ostatnim elementem powyższego algorytmu.

4.1. Piszemy własną funkcję.

Każdy dobrze zaprojektowany program powinien przed zakończeniem swojego działania zadać użytkownikowi pytanie, czy chce on ponownie wykonać przebieg programu czy też nie i w zależności od uzyskanej odpowiedzi podjąć odpowiednią akcję. W języku C istnieje szczególnie wygodny mechanizm do realizacji takiego celu. Są nim funkcje. Dobrze zaprojektowana i przetestowana funkcja może zostać w postaci półskompilowanej włączona do biblioteki kompilatora i używana w każdym nowotworzonym programie.

Przedstawimy teraz funkcję, której zadaniem będzie ustalenie, czy użytkownik programu ma zamiar kontynuować pracę, czy też ją zakończyć. Będzie to funkcja bezargumentowa, która będzie zwracać do miejsca wywołania wartość 1, gdy użytkownik będzie chciał kontynuować pracę, a zero w przeciwnym wypadku. Tekst źródłowy takiej funkcji może wyglądać następująco:

```
/**************************
 *   funkcja  CZYDALEJ    *
 **************************/
czydalej()${
int stat,ods,T,t,N,n;
char u,od,zer,i;
i='1';
od='4';
zer='8';
u="K:";
T=84;  /* kod litery T */
t=116; /* kod litery t */
N=78;  /* kod litery N */
n=110; /* kod litery n */
stat=osen(i,od,zer,u);   /* otwarcie "K:" z iocb 1   */
if(stat!=1)${
   printf("\n\aBlad otwarcia K: ");
   return;
$}    /* koniec if  */
do${  /* poczatek do */
printf("\nCzy chcesz kontynuowac?");
ods=cgetc(i);
if(ods!=T && ods!=t && ods!=N && ods!=n)${
  printf("\n\aOdpowiedz albo T albo N");
  continue;
 $}  /* koniec if */
$}   /* koniec do */
while(ods!=T && ods!=t && ods!=N && ods!=n);
close(i);
return((ods==T || ods==t) ? 1 : 0);
$} /* koniec czydalej() */
```

Funkcja czydalej() działa w następujący sposób:

-wyświetla pytanie: "Czy chcesz kontynuować?" i czeka na naciśnięcie jednego z dwóch klawiszy T lub N;

-gdy zostanie naciśnięty jakikolwiek klawisz, sprawdzane jest czy jest to klawisz T lub N, jeżeli nie jest, to funkcja wyświetla komunikat "Odpowiedz albo T albo N", ponawia pytanie i ponownie czeka na naciśnięcie klawisza;

-gdy naciśnięty zostanie klawisz T (lub małe t), funkcja czydalej() kończy działanie i zwraca do miejsca wywołania wartość 1;

-gdy naciśnięty zostanie klawisz N (lub małe n), funkcja również kończy działanie, zwracając w takiej sytuacji zero do miejsca wywołania.

Zakończenie działania funkcji realizowane jest przy pomocy instrukcji return, która umożliwia przekazanie sterowania w programie z funkcji wywołanej do funkcji wywołującej. Postać tej instrukcji jest następująca:

return(wyrażenie);

Jeśli instrukcja return nie zawiera żadnego wyrażenia, w takiej sytuacji do funkcji wywołującej nie zostanie zwrócona żadna wartość (nastąpi tylko przekazanie sterowania). Gdy instrukcja return zawiera wyrażenie, to wraz z przekazaniem sterowania, do funkcji wywołującej przekazana zostanie wartość, będąca rezultatem funkcji wywołanej. Instrukcja

return((odp==T || odp==t) ? 1: 0);

powoduje wyjście z funkcji czydalej() i jednoczesne przekazanie rezultatu tej funkcji do funkcji wywołującej (main()). Gdy wyrażenie logiczne (odp==T ||odp==t) przyjmie wartość TRUE, to funkcja czydalej() zwróci do miejsca wywołania wartość 1, gdy wyrażenie to przyjmie wartość FALSE - zostanie zwrócone zero.

Uzyskanie tak zwartego zapisu instrukcji return jest możliwe dzięki trójargumentowemu operatorowi warunkowemu "?". Wyrażenia logiczne budowane przy pomocy tego operatora mają ogólną postać:

wyrażenie1 ? wyrażenie2 : wyrażenie3

a ich wartość obliczana jest według następującego algorytmu:

-najpierw obliczana jest wartość wyrażenia1;

-jeśli obliczona wartość jest różna od zera (TRUE), to obliczana jest wartość wyrażenia2 i ona jest wartością wynikową całego wyrażenia logicznego;

-w przeciwnym wypadku wartością wynikową całego wyrażenia jest wartość wyrażenia3.

Operator ? zastępuje konstrukcje logiczne typu:

```
if (y>x)
  z=y;
else
  z=x;
```

dzięki niemu można je zapisać w zwartej postaci:

z=(y>x) ? y : x

Należy zauważyć, że x,y,z mogą być dowolnymi wartościami zmiennych. Zatem wyżej przedstawionej instrukcji można użyć np. do wybrania większej z dwóch liczb.

Kilka słów na temat kompilacji funkcji czydalej(). Przedstawiony wyżej kod źródłowy funkcji należy wprowadzić za pomocą edytora tekstowego do zbioru o nazwie CZYDALEJ.C. Kompilator DEEP BLUE C jej kod półskompilowany zapisze do zbioru o takiej samej nazwie, z rozszerzeniem .CCC. Chcąc dołączyć tę funkcję do każdego nowotworzonego programu, należy przy konsolidacji tego programu umieścić dodatkowy wiersz CZYDALEJ.CCC w zbiorze wskazówek dla linkera.

4.2. Program kalkulator.

Możemy teraz przystąpić do napisania wspomnianego wyżej programu kalkulatora. Oczywiście wykorzystamy w nim funkcję czydalej(). (Nie zapomnij o umieszczeniu wiersza CZYDALEJ.CCC w potrzebnym do konsolidacji tego programu zbiorze wskazówek dla linkera). Oto postać źródłowa tego programu

program 11

```
/*****************************/
/* program11 KALKULATOR      */
/*****************************/
#define   TAK   1
#define   NIE   0
main()
${       /*poczatek funkcji main()  */
 int l1,l2,wynik;
 char oper,zlyoper,znak;
 char arg1[120],arg2[120];
 do     /* poczatek do  */
 ${
  zlyoper=NIE;
  printf("\nWybierz rodzaj dzialania\n");
  printf("+,-,*,/ ");
  oper=getchar();
  znak=getchar(); /* przeskocz RETURN*/

  printf("\nPodaj pierwszy argument: ");
  gets(arg1);
  l1=val(arg1);
  printf("\nPodaj drugi argument: ");
  gets(arg2);
  l2=val(arg2);
  switch(oper)
   ${    /* poczatek switch */
    case '+' : wynik=l1+l2;
               break;
    case '-' : wynik=l1-l2;
               break;
    case '*' : wynik=l1*l2;
               break;
    case '/' : wynik=l1/l2;
               break;
    default  : printf("\n\aWybrales zle dzialanie");
               zlyoper=TAK;
               break;
   $}   /* koniec switch */
  if(zlyoper==NIE)
   printf("\nWynik dzialania: %d\n",wynik);
 $}     /* koniec do   */
 while(czydalej());
$}   /* koniec main()  */
```

Jest to pierwszy większy program i zasługuje na specjalne

wyjaśnienie. Trzy linie:

```
printf("\nWybierz rodzaj działania\n");
printf("+,-,*,/");
oper = getchar();
```

informują o działaniach, jakie program może wykonać i wczytują z klawiatury znak, odpowiadający wybranej operacji do zmiennej oper. Linie:

```
printf("\nPodaj pierwszy argument: ");
gets(arg1);
l1=val(arg1);
printf("\nPodaj drugi argument: ");
gets(arg2);
l2=val(arg2);
```

wprowadzają z klawiatury wartości argumentów, na których ma być wykonana wybrana operacja. Wprowadzanie argumentów realizowane jest dwustopowo. Najpierw wczytywany jest łańcuch znaków stanowiących argument (funkcja gets()), a następnie łańcuch ten zamieniany jest na liczbę za pomocą funkcji val(). Funkcja ta działa podobnie, jak instrukcja VAL z BASIC-a. W tym miejscu można zadać pytanie: czy nie można wprowadzić z klawiatury ciągu cyfr, dokonując automatycznie konwersji tego ciągu na odpowiadającą mu liczbę. Niestety, implementacja DOCC nie zawiera takiej możliwości, aczkolwiek w standardowym języku C istnieje funkcja realizująca takie zadanie (scanf()). Następnie za pomocą instrukcji switch wybierany jest rodzaj operacji, która ma być wykonana i następuje jej realizacja.

4.3. Instrukcja switch.

W programie "kalkulator" rodzaj akcji, którą program ma podjąć zależy od rodzaju działania, który wybiera użytkownik. W oparciu o informacje dotyczące instrukcji if i else fragment programu realizujący ten wybór można byłoby zapisać w poniższy sposób:

```
if (oper=='+')
  wynik = l1+l2;
else if (oper=='-')
  wynik = l1-l2;
else if (oper=='*');
  wynik = l1*l2;
else if (oper=='/')
  wynik = l1/l2;
else {
printf("wybrales zle dzialanie\n");
zlyoper = TAK }
```

Instrukcja switch pozwala uprościć zapis i uczynić go bardziej czytelnym.

Konstrukcję zbudowaną z instrukcji switch rozpoczyna słowo kluczowe switch, po którym występuje wyrażenie (lub zmienna) ujęte w nawiasy (..). Następnie w nawiasach {..} umieszczona jest lista wszystkich możliwych przypadków (wyborów), od których zależy rodzaj podejmowanego przez program działania. Każdy przypadek definiowany jest za pomocą instrukcji case, za którą, po dwukropku, definiowane jest działanie, które musi być podjęte przez program, gdy zaistnieje zdefiniowany przypadek. Ciąg instrukcji definiujących działanie programu w zaistniałym przypadku musi być zakończony instrukcją break lub nawiasem zamkającym "}" definiowanie wszystkich przypadków. Na przykład:

```
case '+' :  wynik = l1+l2
            break;
```

Instrukcja break powoduje, że gdy wykonana zostanie akcja związana z zaistnieniem określonego przypadku, nie będą sprawdzane przypadki zdefiniowane za nią i dalsze wykonywanie programu rozpocznie się od instrukcji umieszczonej po nawiasie zamykającym "}" całą konstrukcję switch.

Słowo kluczowe default służy do definiowania działania, które program podejmuje w sytuacji, gdy nie zostanie spełniony żaden z przypadków zdefiniowanych instrukcjami case. Zatem z programu "kalkulator", gdy użytkownik wprowadzi niedopuszczalny rodzaj działania, wykonane zostaną instrukcje:

```
printf("wybrales zle dzialanie\n");
zlyoper = TAK;
```

i nie zostanie wyświetlony wynik działania. Wyświetlenie wyniku działania uzależnione jest od wartości wyrażenia (zlyoper==NIE). Jeśli wyrażenie to przyjmie wartość różną od zera (nie zostanie wybrany niedozwolony rodzaj działania), wynik zostanie wyświetlony, w przeciwnym wypadku wyświetlanie wyniku jest pominięte.

4.4. Instrukcja continue.

Wnikliwy programista, analizując program "kalkulator", dostrzeże zapewne nieeleganckie rozwiązanie w nim zawarte. Polega ono na tym, że w sytuacji, gdy użytkownik wybierze niedozwolony rodzaj operacji (inny niż +,-,*,/) program wykona zupełnie niepotrzebnie całą pętlę. Znacznie lepiej byłoby, gdyby w takiej sytuacji program zażądał dokonania ponownego wyboru, natychmiast po stwierdzeniu wybrania niedozwolonej operacji. Poprawmy zatem ten program, aby wyeliminować tę wadę. W tym celu użyjemy instrukcji continue. Oto poprawiona wersja programu:

Program 12

```
/*****************************/
/* program12 KALKULATOR      */
/* wersja z inst. continue   */
/*****************************/
#define   TAK   1
#define   NIE   0
main()
{       /*poczatek funkcji main()  */
 int l1,l2,wynik;
 char oper,zlyoper,znak;
 char ars1[120],ars2[120];
 do     /* poczatek do  */
 {
  printf("\nWybierz rodzaj działania\n");
  printf("+,-,*,/ ");
  oper=getchar();
  znak=getchar(); /* przeskocz RETURN*/

  /* Teraz sprawdz, czy wprowadzono dozwolony rodzaj działania */
  /* i ewentualnie zakonczenie wyboru, gdy wybrano działanie niedozw.  */
  if(oper!='+' && oper!='-' && oper!='*' && oper!='/')
    {
     printf("Wybrales zle działanie\n");
     continue;
    }
  /* od tego miejsca rozpocznij wykonywanie programu   */
  /* gdy wybrano dopuszczalne działanie                */
  printf("\nPodaj pierwszy argument: ");
  gets(ars1);
  l1=val(ars1);
  printf("\nPodaj drugi argument: ");
  gets(ars2);
  l2=val(ars2);
  if(l2==0) break;
  /* zatrzymaj program gdy drugi argument jest zerem   */
  switch(oper)
   {     /* poczatek switch */
    case '+' : wynik=l1+l2;
               break;
    case '-' : wynik=l1-l2;
               break;
    case '*' : wynik=l1*l2;
               break;
    case '/' : wynik=l1/l2;
               break;
   }   /* koniec switch */
   printf("\nWynik działania: %d\n",wynik);
 }     /* koniec do   */
 while(czydalej());
}    /* koniec main()  */
```

Dzięki użyciu instrukcji continue program ten po stwierdzeniu, że użytkownik wybrał niedozwolony rodzaj operacji, wyświetli komunikat "Wybrales zle działanie" i nakaże dokonać

ponownie wyboru.

poniższy schemat ilustruje działanie instrukcji continue:

```
do
{      /* początek pętli */
   if (wyrażenie logiczne1)
     {
     .....
     continue;
     }

      /* pozostała część pętli */
     }
while (wyrażenie logiczne2);
```

Wykonywany jest początek pętli do i^ obliczana jest wartość wyrażenia logicznego1. Gdy jest ona różna od zera, wykonywane są instrukcje umieszczone po nim, na których końcu znajduje się instrukcja continue. Jej wykonanie powoduje przeniesienie sterowania do instrukcji while, a stąd, gdy spełniony jest warunek zdefiniowany wyrażeniem logicznym2, następuje skok na początek pętli. W tej sytuacji pozostała część pętli (w programie "kalkulator" zawiera ona instrukcję switch) zostanie wykonana jedynie wówczas, gdy wyrażenie logiczne1 przyjmie wartość zero (FALSE). Należy zauważyć, że w takiej sytuacji nie ma potrzeby używania instrukcji default.

4.5. Instrukcja break.

W programie 11 zastosowano instrukcję break do "przeskoczenia" pozostałych do zbadania przypadków zdefiniowanych w instrukcji wyboru switch. Natomiast w programie 12 instrukcja ta została użyta w celu opuszczenia pętli programowej:

```
if(l2==0) break;
```

Jeżeli założymy, że użytkownik wprowadził drugi argument działania równy zero, to powyższa instrukcja spowoduje opuszczenie pętli i wykonanie instrukcji występującej po while, które zamyka całą pętlę. Ogólnie, sposób działania instrukcji break można zilustrować schematem:

```
do
{    /* początek pętli  */
....
if (warunek) break;
       /* pozostała część pętli */
}
while(warunek);
      /* koniec pętli */
```

Takie zastosowanie instrukcji break bywa czasami bardzo przydatne. Na przykład: w programie 11 brak było badania wartości drugiego argumentu wykonywanej operacji; zakładając, że wartość ta była zerem i wykonywana była operacja dzielenia, wystąpiłby w takiej sytuacji błąd wykonania programu (próba dzielenia przez zero). W programie 12 przewidziano taką ewentualność i za pomocą instrukcji break pominięto wykonywanie działania. Oczywiście, że zamiast zakończenia w takiej sytuacji działania programu należałoby napisać sekwencję instrukcji wczytujących ponownie wartość drugiego argumentu.

4.6. Instrukcja for.

Instrukcja for jest prawdopodobnie najczęściej używaną instrukcją języka C, przeznaczoną do konstruowania pętli programowych. Najważniejsza różnica pomiędzy pętlą for a przedstawionymi wcześniej pętlami while i do..while polega na tym, że te ostatnie stosuje się, gdy dokładnie nie wiadomo, ile razy mają być wykonane sekwencje instrukcji w nich zawarte. Natomiast instrukcja for jest bardzo pożyteczna do tworzenia pętli, o których z góry wiadomo, że mają być wykonane ściśle określoną liczbę razy. Działanie tej instrukcji zilustrujemy programem obliczającym kwadraty kolejnych liczb naturalnych, np.: kwadraty liczb od 1 do 100. Oto postać takiego programu:

Program 13

```
main()
{
int liczba;
char c;
printf("\Kwadraty liczb naturalnych\n\n");
for(liczba=1;liczba<=100;liczba=liczba+1)
  {
  printf("\nliczba: %d kwadrat %d",liczba,liczba*liczba);
  }
c=getchar();
}
```

Instrukcja for zawiera ujęte w nawiasy (...) trzy części, oddzielone od siebie średnikami. Część pierwsza (liczba=1) jest częścią inicjującą pętli, wykonywana jest ona raz na początku pętli i ustawia wartość początkową licznika pętli. Część druga, zwana częścią sterującą pętli, (liczba<=100) jest wyrażeniem logicznym, od wartości którego zależy czy pętla zostanie wykonana ponownie, czy nastąpi jej opuszczenie. Pętla jest wykonywana tak długo, jak długo wyrażenie to przyjmuje wartość logiczną TRUE. Trzecią część pętli (liczba=liczba+1) definiuje krok, z jakim zmieniana jest wartość licznika pętli. Część ta nazywana jest również częścią modyfikującą pętli. Po instrukcji for umieszczone są, ujęte w nawiasy {...} instrukcje stanowiące pętlę. Jeżeli pętla składa się tylko z jednej instrukcji, to nawiasy { i } można pominąć.

Nietrudno dostrzec, że program 13 można skonstruować również używając pętli while. Będzie on miał wówczas taką postać:

Program 14

```
main()
{
int liczba;
char c;
printf("\Kwadraty liczb naturalnych\n\n");
liczba=1;     /* część inicjująca pętli */
while(liczba<=100)    /* część sterująca */
  {
  printf("\nliczba: %d kwadrat %d",liczba,liczba*liczba);
  liczba=liczba+1;    /* część modyfikująca */
  }
c=getchar();
}
```

Efekty działania programów 13 i 14 są identyczne. Którą zatem konstrukcję należy stosować w tworzonych programach? Nie ma jednoznacznej odpowiedzi na takie pytanie. Obie konstrukcje są równoważne. Pętla for jest bardziej przejrzysta, gdyż skupia wszystkie elementy sterujące jej wykonaniem w jednym miejscu programu.

Dowolną z trzech części instrukcji for można pominąć, jednak oddzielający je średnik musi pozostać. Pominięcie części inicjującej i modyfikującej jest równoznaczne z usunięciem odpowiednich instrukcji z programu 14. (Można przeanalizować, jak ten program będzie działał w takiej sytuacji). Pominięcie części sterującej jest równoznaczne założeniu, że definiujące ją wyrażenie logiczne zawsze przyjmuje logiczną wartość TRUE. Korzystając z tej cechy, można za pomocą instrukcji for skonstruować tzw. wieczną pętlę: for(;;).

Z reguły wyjście z takiej pętli następuje przy pomocy instrukcji continue, break lub return. Jednak taka konstrukcja bywa czasami bardzo przydatna.

Omawiając instrukcję for, warto przy okazji przedstawić dwa jednoargumentowe operatory dodawania i odejmowania, bardzo wygodne w użyciu i stosowane najczęściej do modyfikowania wartości licznika pętli. Na przykład: instrukcję przypisania

```
liczba=liczba+1;
```

można w języku C zastąpić prostym wyrażeniem: liczba++. Zatem instrukcji for z programu 13 można nadać taką postać:

```
for(liczba=1;liczba<=100;liczba++)
```

Operator "++" realizuje działanie: "zwiększ o jeden wartość zmiennej". Operator ten może być również użyty w instrukcji przypisania:

```
j=i++;
```

Instrukcja powyższa zostanie wykonana w następujący sposób:

-wartość zmiennej i przypisana jest do zmiennej j i jest zwiększana o jeden. Zatem instrukcja j=i++ jest równoważna dwom instrukcjom:

```
j=i;
i=i+1;
```

Można budować również takie instrukcje:

```
j=++i;
```

są one wykonywane nieco inaczej: najpierw wartość zmiennej i jest zwiększana o jeden, a potem zmiennej j przypisywana jest wartość zmiennej i. Zatem odpowiada to instrukcjom:

```
i=i+1;
j=i;
```

Analogicznie działa jednoargumentowy operator odejmowania "--". Instrukcja j=i--; jest równoważna instrukcjom:

```
j=i;
i=i-1;
```

natomiast instrukcja j=--i; zastępuje:

```
i=i-1;
j=i;
```

Operatory "++" i "--" mogą być użyte w złożonych wyrażeniach arytmetycznych, jak na przykład:

```
j=i++-k+3;
```

które zastępuje poniższe instrukcje:

```
j=i-k+3;
i=i+1;
```

Uważny czytelnik, który wprowadził i skompilował program "kalkulator", mógł zauważyć, że gdy dodane (lub pomnożone) zostaną dwie duże liczby np.: 10000 i 30000, to program wyprowadza niepoprawne wyniki. Również błędne wyniki będzie generował program 13, gdy podnoszone do kwadratu liczby będą większe od 182. Co jest tego przyczyną? Dzieje się tak, ponieważ w pierwszym jak i w drugim programie mnożone, dodawane, dzielone i odejmowane liczby zostały zadeklarowane jako liczby całkowite (typu int). Na wstępie zaznaczono, że w implementacji DBC liczba typu int może przyjmować wartości z przedziału [-32768,32767]. Jeżeli w wyniku wykonania operacji arytmetycznej powstaje rezultat nie mieszczący się w tym przedziale, następuje przepełnienie (tzn. wynik przekracza zakres liczby, którą można zapisać na 15 bitach, bit 16 jest bitem znaku), które powoduje, że wyprowadzane wyniki są błędne. Cóż zatem robić w takiej sytuacji? Otóż, standardowy język C dopuszcza deklaracje zmiennych typu float i double, których zakres jest nieporównywalnie większy od zmiennych typu int. Aczkolwiek DBC nie posiada zaimplementowanych tego typu danych, to zawiera bogatą bibliotekę funkcji przeznaczonych do realizacji arytmetyki zmiennoprzecinkowej. Ze względu na liczbę funkcji zawartych w tej bibliotece, ich opis oraz sposób wykorzystania, ilustrowany odpowiednimi programami, zostanie przedstawiony w następnym wydaniu specjalnym IKS-a.

5. TABLICE I WSKAŹNIKI

W języku C tablicami są zmienne przeznaczone do przechowywania wielu wartości tego samego typu, do których można odwoływać się poprzez jedną, wspólną dla wszystkich tych wartości nazwę. Elementy tablic numerowane są od zera. Kompilator DBC dopuszcza stosowanie w programach dwóch typów tablic: tablic znakowych (ich elementami są zmienne typy char) i tablic, których elementami są zmienne typu int. Zarówno pierwsze jak i drugie tablice mogą być jednowymiarowe. Tablice, podobnie jak zmienne, muszą być zadeklarowane przed ich użyciem. Deklaracja tablicy składa się z określenia typu tablicy (int,char), jej nazwy i rozmiaru podanego w nawiasach kwadratowych [..]. Na przykład:

```
char tab[9];
```

jest deklaracją tablicy znakowej o nazwie "tab", zawierającej 9 elementów [0,1,...8], a deklaracja:

```
int liczby[9];
```

jest deklaracją tablicy 9 elementowej o nazwie liczby, której elementami są zmienne typu int (liczby całkowite). Należy pamiętać, że obszar pamięci zajęowany przez tablicę typu int jest dwa razy większy od liczby jej elementów (każda zmienna typu int zajmuje dwa bajty pamięci). Odwołanie do określonego elementu tablicy realizowane jest poprzez nazwę tablicy i numer (indeks) tego elementu ujęty w nawiasy [..]. Ponieważ elementy tablic numerowane są od zera, to indeks o wartości zero wskazuje pierwszy element, indeks 1 - drugi element, indeks 2 - trzeci, itd. Zatem instrukcja:

```
liczby[2]=2;
```

przypisuje trzeciemu elementowi tablicy o nazwie "liczby" wartość 2. Natomiast instrukcja: liczby[9]=14; jest błędną w języku C, ponieważ element o indeksie 9 fizycznie nie istnieje. Tego typu błędy w programie są bardzo trudne do wykrycia, tym bardziej, że żaden kompilator języka C, DBC również, błędów takich nie sygnalizuje podczas kompilacji programu. Tylko nieliczne programy (np. program Lint systemu UNIX), napisane specjalnie w celu kontrolowania poprawności składni kodu źródłowego programu w C, są w stanie błędy takie wykryć. Kontrola "wzrokowa" tekstu źródłowego programu, też z reguły takich błędów nie wykrywa - wszystko wygląda poprawnie: tablica jest 9 elementowa, operacja jest wykonana na elemencie o indeksie 9 a skompilowany program nie działa poprawnie.

5.1. Tablice znakowe.

Ponieważ DBC dopuszcza jedynie stosowanie tablic jednowymiarowych, to tablice znakowe są traktowane jako łańcuchy znaków lub krótko - łańcuchy. Kompilator DBC oznacza, w wewnętrznej reprezentacji, koniec łańcucha znakiem, którego kod ATASCII jest równy zeru (serduszko). Pisząc własne funkcje operujące na łańcuchach, należy ten znak (reprezentacja oktalna '\0') traktować jako wskaźnik końca łańcucha.

Każdy, kto pisał programy w języku BASIC wie, że przypisanie wartości zmiennej tekstowej może być wykonane za

pomocą instrukcji:

```
ALA$="WITAJ"
```

Natomiast w języku C instrukcja przypisania:

```
tab="okno";
```

jest niedopuszczalna (tab - tablica znakowa). W DBC wprowadzenie wartości do elementów tablicy znakowej wymaga albo zdefiniowania wskaźnika (będzie o nich mowa trochę później) do tej tablicy albo wykonania szeregu instrukcji przypisania, z których każda przypisze określoną wartość określonemu elementowi tablicy. W ogóle język C nie posiada instrukcji operujących na łańcuchach jako całości. Na pierwszy rzut oka wydaje się to być niewygodne. Jednak operując wskaźnikami, problem ten można doskonale rozwiązać.

5.2. Klasyczne podejście do przetwarzania tablic.

Tablice w języku C można przetwarzać, podobnie jak w innych językach wysokopoziomowych, odwołując się do ich elementów poprzez indeksy. Taką metodę przetwarzania ilustrują przedstawione niżej dwa programy.

Program 15

```
/**********************************
 * program15 PRZESTAWIANKA        *
 **********************************
 */
main()${
char znak,wyraz[128];
int i,j,dl;
 do${
 printf("\nNapisz dowolny wyraz:\n");
 dl=gets(wyraz);
 if(dl<=0) continue;
 printf("\nNapisales:\n");
   for(i=0;i<=dl-1;i++)
        printf(" %c",wyraz[i]);
/* teraz przestaw znaki we wprowadzonym wyrazie */
   for(i=0,j=dl-1;i<j;i++,j--)${
     znak=wyraz[i];
     wyraz[i]=wyraz[j];
     wyraz[j]=znak;
    $)   /* koniec petli for  */
/* wyswietl przestawiony wyraz   */
 printf("\nA tak to wygląda po przestawieniu\n");
   for(i=0;i<=dl-1;i++)
     printf(" %c",wyraz[i]);
 $)   /* koniec do  */
 while(czydalej());
$)  /* koniec main  */
```

Program 15 wykonuje przestawienie znaków w łańcuchu, w taki sposób, że pierwszy znak staje się ostatnim, i drugi przedostatnim, itd. Łańcuch znaków po wczytaniu z klawiatury jest wyświetlany w normalnej postaci i w postaci przestawionej. Tekst źródłowy tego programu jest na tyle prosty, że nie wymaga szczegółowego komentarza. Na uwagę zasługuje jedynie instrukcja for organizująca pętlę, w której dokonywane jest przestawienie znaków w łańcuchu:

```
for(i=0,j=dl-1;i<j;i++,j--)
```

Instrukcja ta steruje jednocześnie dwoma indeksami do tablicy wyraz[]. Indeks i "zaczyna" po tablicy od jej początku do końca, natomiast indeks j od końca do początku. Taka konstrukcja w języku C jest możliwa, dzięki operatorowi przecinkowemu. Znak "," jest operatorem jedynie wówczas, gdy występuje pomiędzy wyrażeniami, np.:

```
i=0,j=0,k=0,l=0;
```

jest instrukcją ustawiającą wartości czterech zmiennych: i, j, k, l na zero. Operatora przecinkowego można używać również do tworzenia takich instrukcji:

```
i=5,i+2;
```

po jej wykonaniu zmienna i ma wartość 7. Przecinek nie pełni funkcji operatora, gdy oddziela zmienne w deklaracjach lub argumenty w wywołaniu funkcji.

Program 16.

```
/**********************************
 * program16 Sortowanie tablicy   *
 *      liczb calkowitych         *
 *   wedlug algorytmu Shell'a     *
 **********************************
 */
#define   dl  10
main()${
int tab[dl],i,j,k,krok;
char liczba[6];
do${
 /* wczytaj wartosci do tab  */
 for(i=0;i<=dl-1;i++)${
  printf("\nPodaj %d element",i);
  gets(liczba);
  tab[i]=val(liczba);
 $)   /* for */
 /* teraz wyswietl wszystkie elementy
  * w takiej kolejnosci w jakiej zostaly
  * wczytane
  */
 printf("\nwprowadziles nastepujace liczby:\n");
 for(i=0;i<=dl-1;i++)
   printf(" %d",tab[i]);
 /* teraz uporzadkuj rosnaco tablice */
 for(krok=dl/2;krok>0;krok/=2)
   for(i=krok;i<dl;i++)
     for(j=i-krok;j>=0 && tab[j]>tab[j+krok];j-=krok)${
       k=tab[j];
       tab[j]=tab[j+krok];
       tab[j+krok]=k;
     $)  /* for */
 /* teraz wyswietl posortowana tablice */
 printf("\n\ntablica posortowana\n");
  for(i=0;i<=dl-1;i++)
    printf(" %d",tab[i]);
 $) /* do */
while(czydalej());
$) /* main */
```

Program 16 porządkuje 10 elementową tablicę liczb. Najpierw wprowadza z klawiatury wartości do elementów tej tablicy, wyświetla je, porządkuje metodą Shell'a elementy od najmniejszego do największego i wyświetla elementy uporządkowane.

Oba programy wykorzystują przedstawioną wcześniej funkcję czydalej(). Przyjrzyjmy się jej jeszcze raz, ponieważ zastosowano w niej nowy typ zmiennej - zmienną wskaźnikową.

5.3. Wskaźniki.

Zmienne wskaźnikowe, lub krótko wskaźniki, podobnie jak inne zmienne w języku C muszą być zadeklarowane przed ich użyciem. Każdy wskaźnik posiada typ związany z obiektem, który wskazuje. Deklaracja:

```
char *u
```

nakazuje kompilatorowi DBC utworzenie wskaźnika typu znakowego, który wskazywał będzie obiekty tego samego typu, tzn. zmienne i tablice znakowe typu char. Natomiast deklaracja:

```
int *i
```

nakazuje kompilatorowi utworzenie wskaźnika typu integer (wskazywł on będzie zmienne i tablice tego samego typu). DBC nie dopuszcza deklaracji wskaźników innych typów.

W języku C wskaźnik jest zmienną zawierającą adres obiektu (zmiennej lub elementu tablicy). Na przykład: jeśli w programie zadeklarowaliśmy tablicę typu integer:

```
int tab[10];
```

oraz wskaźnik:

```
int *wsk
```

to sam fakt deklaracji nie wiąże jeszcze wskaźnika ze wskazywanym przez niego obiektem (np. zadeklarowaną wyżej tablicą). Jeśli chcemy, aby wskaźnik został związany z jakąkolwiek zmienną musimy przypisać adres tej zmiennej do tego wskaźnika. Przypisanie adresu dokonywane jest przez operator adresu "&". Na przykład: instrukcja:

```
wsk=&tab[0];
```

powoduje, że wskaźnikowi przypisany zostanie adres pierwszego elementu tablicy tab. Jeżeli teraz takiego wskaźnika użyjemy w instrukcji:

```
element=*wsk;
```

to zmienna element zawierać będzie pierwszy element tablicy tab. Operator "*" powoduje, że jego argument jest traktowany jako adres zmiennej, której zawartość ma być użyta w instrukcji, w której operator ten występuje. Łatwo zauważyć, że instrukcje:

```
wsk=&tab[i];
element=*wsk;
```

są równoważne jednej:

```
element=tab[i];
```

gdzie zmienna element jest zadeklarowaną wcześniej zmienną typu integer. Po cóż zatem wskaźniki w języku C, skoro można bez ich pomocy zrealizować potrzebną operację?

Najistotniejszym powodem, dla którgo konstruktorzy języka C wyposażyli go w mechanizm wskaźników, jest sposób realizacji funkcji i przekazywania do nich argumentów.

Argumenty w języku C przekazywane są do wywoływanych funkcji poprzez wartość. Chcąc zatem przekazać do funkcji, jako argument np. 100 elementową tablicę, należałoby z każdego elemenetu tej tablicy uczynić osobny argument. Byłoby to bardzo niewygodne. Posługując się wskaźnikami, można przekazać funkcji, jako argument, adres początku tej tablicy, zawarty we wskaźniku. Działanie tego mechanizmu pokażemy na przykładzie programu 15. Napiszemy ten program w innej postaci, tworząc funkcję odwroc(), której zadaniem będzie przestawienie znaków w łańcuchu. Argumentem tej funkcji będzie adres łańcucha i jego długość. Oto postać tej funkcji.

Funkcja 2

```
/**********************
 * funkcja ODWROC()   *
 **********************/
odwroc(wyraz,dlugosc)
char *wyraz;
int dlugosc;
${ /* poczatek funkcji */
char znak;
char *koniec;
dlugosc-=1;
koniec=wyraz+dlugosc;
for(;wyraz<koniec;wyraz++,koniec--)${
 znak=*wyraz;
 *wyraz=*koniec;
 *koniec=znak;
 $} /* for */
return;
$} /* koniec odwroc() */
```

Tekst źródłowy tej funkcji należy wprowadzić do zbioru o nazwie ODWROC.C. Kompilator DBC po jej skompilowaniu, zapisze kod półskompilowany do zbioru ODWROC.CCC. Chcąc użyć tej funkcji w programie 15, należy go zmodyfikować, zastępując pętlę, w której realizowane było przestawianie znaków łańcucha wywołaniem funkcji odwroc(). Wersja zmodyfikowana tego programu ma postać:

Program 17

```
/*********************************
 * program17 PRZESTAWIANKA       *
 * wykorzystuje funkcje odwroc() *
 * do przestawiania znakow w     *
 * lancuchu                      *
 *********************************
 */
main()${
char znak,wyraz[120];
int i,j,dl;
 do${
 printf("\nNapisz dowolny wyraz:\n");
 dl=gets(wyraz);
 if(dl<=0) continue;
 printf("\nNapisales:\n");
   for(i=0;i<=dl-1;i++)
        printf(" %c",wyraz[i]);
/* teraz przestaw znaki we wprowadzonym wyrazie */
 odwroc(wyraz,dl);
/* wyswietl przestawiony wyraz   */
 printf("\nA tak to wyglada po przestawieniu:\n");
   for(i=0;i<=dl-1;i++)
      printf(" %c",wyraz[i]);
  $)   /* koniec do  */
 while(czydalej());
$)  /* koniec main  */
```

Przed konsolidacją programu 17 należy wprowadzić do zbioru wskazówek dla linkera dodatkowy wiersz (ODWROC.CCC).

Efekty działania programów 15 i 17 są identyczne.

Przedstawimy teraz dokładnie, jak funkcjonują wskaźniki w funkcji odwroc() i w jaki sposób przekazywane są do niej argumenty. Instrukcja odwroc(wyraz,dl); w programie 17 stanowi wywołanie funkcji odwroc(). Funkcja ta otrzymuje dwa argumenty:

-nazwę tablicy "wyraz"z funkcji main();

-liczbę elementów tablicy (zmienna dl)

Wewnątrz funkcji odwroc() deklaracja char *wyraz; tworzy wskaźnik do tej tablicy. Zbieżność nazw tablicy i zmiennej wskaźnikowej nie jest przypadkowa. Gdybyśmy użyli wskaźnika o innej nazwie, np. char *poczatek; to jego wartość po takiej deklaracji byłaby nieokreślona.

Wewnątrz funkcji odwroc() mamy zadeklarowane dwie zmienne: char znak; i char *koniec;. Zmienna "znak" jest zwykłą zmienną roboczą, wykorzystywaną do chwilowego przechowania przestawianych elementów tablicy. Natomiast wskaźnik "koniec" jest "prywatnym" wskaźnikiem funkcji odwroc() i wskazuje ostatni element przestawianej tablicy. Instrukcja:

```
koniec=wyraz+dlugosc;
```

inicjuje wartość wskaźnika "koniec", przypisując mu adres ostatniego elementu tablicy "wyraz" (do adresu początkowego tablicy zawartego we wskaźniku "wyraz" dodawana jest liczba elementów tablicy). Należy w tym miejscu zwrócić uwagę na jeden aspekt takiego inicjowania wartości wskaźników. Powyższe inicjowanie jest poprawne, gdyż tablica "wyraz" jest tablicą znakową (każdy jej element zajmuje jeden bajt pamięci). Natomiast gdyby tablica "wyraz" była typu int (każdy element zajmuje dwa bajty pamięci), to poprzednia inicjacja wskaźnika "koniec" byłaby błędna. Należałoby wówczas użyć takiej:

```
koniec=wyraz+(2*dlugosc);
```

Przestawianie elementów tablicy realizowane jest w pętli for(..). Pętla ta nie posiada części inicjującej, ponieważ nie ma potrzeby definiowania takiej części. Część sterującą pętli stanowi wyrażenie:

```
wyraz<koniec;
```

będące porównaniem wartości dwóch wskaźników (adresów). Pętla będzie wykonywana tak długo, dopóki wartość wskaźnika wskazującego na początku pętli pierwszy element tablicy "wyraz" będzie mniejsza od wartości wskaźnika wskazującego ostatni element.

Część modyfikującą pętli stanowią wyrażenia: wyraz++, koniec--. Oznacza to, że po każdorazowym wykonaniu pętli wskaźnik początku tablicy zwiększany jest w taki sposób, aby wskazywał on następny element, a wskaźnik końca - zmniejszany w taki sposób, aby wskazywał poprzedzający element. Jak widać, na wskaźnikach można wykonywać operacje arytmetyczne. Wskaźniki są

adresami, zatem wykonywanie na nich operacji dodawania lub odejmowania jest zwiększaniem lub zmniejszaniem adresu pamięci. Można również dodawać do siebie (lub odejmować) wartości wskaźników. Jednak zawsze należy pamiętać, aby poddawane takim operacjom wskaźniki były tego samego typu. Dodanie wartości wskaźnika typu znakowego do wartości wskaźnika typu int i użycie tak wyznaczonego adresu do jakiejkolwiek operacji może spowodować nieobliczalne skutki w programie. Z czego to wynika? Deklarując wskaźnik, np. char *znak; informujemy kompilator, że wskaźnik ten wskazywał będzie zmienne znakowe. Jeżeli na takim wskaźniku wykonamy operację znak++, to jego wartość zostanie zwiększona o jeden i tak jest poprawnie. Natomiast, gdyby wskaźnik ten został zadeklarowany jako typu int, to operacja znak++ zwiększyłaby jego wartość o dwa (nie o jeden), gdyż po niej ma on wskazywać następny obiekt typu int. Jest to logiczne, gdyż zmienne typu int zajmują dwa bajty pamięci.

Należy zwrócić również uwagę na deklarację argumentów funkcji odwroc() wewnątrz funkcji. Jeżeli funkcja otrzymuje jakieś argumenty, to muszą być one zadeklarowane tuż po nazwie funkcji (poza nawiasami $(..$). Wszystkie zmienne deklarowane wewnątrz nawiasów $(..$) są prywatnymi zmiennymi funkcji, w której zostały zadeklarowane i żadna inna funkcja, w tym również main() nie ma do nich dostępu.

Pozostańmy jednak nadal przy wskaźnikach. Jak zaznaczono wcześniej, sam fakt zadeklarowania wskaźnika nie powoduje, że wskazuje on obiekt programu. Aby wskaźnik można było sensownie wykorzystać, musi on zostać prawidłowo zainicjowany, tzn. musi mu zostać przypisana wartość, będąca adresem jednej ze zmiennych programu. Używanie wskaźników bez nadania im wartości początkowej (inicjacji) jest częstym błędem popełnianym przez początkujących programistów w języku C. Przed opisaniem metod inicjowania wartości wskaźników należy zwrócić uwagę na jeszcze jeden aspekt ich stosowania. W języku C stanowią one alternatywną metodę przetwarzania dużych obszarów danych, zapisanych w pamięci komputera. Alternatywną, gdyż jak wynika z przedstawionych wcześniej przykładów, prawie wszystko to, co można wykonać w programie przy ich zastosowaniu, można również zrealizować stosując metodę indeksowania elementów tych obszarów. Często stosowanie indeksów bywa wygodniejsze, a często jest odwrotnie.

Kompilator DBC realizując deklarację zmiennej wskaźnikowej nie sprawdza, czy został zadeklarowany obiekt programu, z którym ten wskaźnik ma być skojarzony. Jest to kłopot programisty. Stąd tak istotne znaczenie ma inicjowanie wartości wskaźników. Sposoby inicjowania wskaźników pokażemy na niżej przedstawionych przykładach:

Niech deklaracja

```
char tablica[100];
```

będzie deklaracją tablicy znakowej, a

```
char *wsk;
```

deklaracją wskaźnika tego samego typu. Instrukcja przypisania:

```
wsk=&tablica[0];
```

inicjuje wskaźnik wsk na pierwszy (początkowy) element tablicy. Poprawne jest również zainicjowanie wskaźnika w taki sposób:

```
wsk=&tablica[i];
```

wskazywał on wówczas będzie i-ty element tablicy. Wskaźnik zainicjować można również w poniższy sposób:

```
wsk="to jest tekst";
```

Tak zainicjowany wskaźnik zawiera adres obszaru pamięci, zawierającego łańcuch "to jest tekst". Jeżeli tak zainicjowanego wskaźnika użyjemy w instrukcji przypisania:

```
znak=*(wsk+2);
```

gdzie znak jest zmienną znakową, to zawierać ona będzie kod trzeciego znaku z tego łańcucha, tj. kod spacji (=20).

Wskaźnik może być zainicjowany na początkowy element tablicy poprzez instrukcję:

```
wsk=tablica;
```

Z ostatniego sposobu inicjowania wynika, że nazwa tablicy bez nawiasów [..] traktowana jest jak wskaźnik. Tak jest w istocie. W języku C każde odwołanie do elementu tablicy realizowane jest poprzez wskaźnik do jej początku. Czy można zatem traktować jako równoważne deklaracje wskaźników i funkcji, np.:?

```
char tablica[100];

char *tablica;
```

Niestety, nie. Deklaracja tablicy musi zawsze wystąpić wszędzie tam, gdzie konieczne jest zarezerwowanie pewnego obszaru pamięci na dane programu. Deklaracja wskaźnika problemu tego nie rozwiązuje. Natomiast jeżeli chodzi o metodę dostępu do elementów tych danych, to oba podejścia, tj. klasyczne, oparte na indeksowaniu elementów, jak i na wskaźnikach - są równorzędne.

Istnieje tylko jedna sytuacja, w której powyższe deklaracje są równoważne. Ma ona miejsce wówczas, gdy dotyczą one parametrów formalnych funkcji. Można to zilustrować na przykładzie prezentowanej już funkcji odwróć(), której jednym z argumentów jest tablica wyraz[]. Wewnątrz funkcji jest ona zadeklarowana jako wskaźnik:

```
odwroc(wyraz,dlugosc)
char *wyraz;
```

i jej elementy przetwarzane są przy zastosowaniu wskaźników. Tablicę wyraz[] można zadeklarować wewnątrz funkcji również tak:

```
odwroc(wyraz,dlugosc);
char wyraz[];
```

i wówczas jej elementy powinny być przetwarzane przy zastosowaniu metody klasycznej, tj. opartej na indeksach. Funkcja odwroc() powinna mieć w tym przypadku taką postać:

```
/**************************
 * funkcja ODWROC()       *
 * wersja "indeksowa"     *
 **************************/
odwroc(wyraz,dlugosc)
int dlugosc;
char wyraz[dlugosc];
${ /* poczatek funkcji */
char znak;
int i,j;
dlugosc-=1;
for(i=0,j=dlugosc;i<j;i++,j--)${
 znak=wyraz[i];
 wyraz[i]=wyraz[j];
 wyraz[j]=znak;
 $} /* for */
return;
$} /* koniec odwroc() */
```

DODATEK A

"BIBLIOTEKA DEEP BLUE C"

A1. Wstęp.

W przeciwieństwie do innych języków C nie posiada na stałe wbudowanych instrukcji we/wy. Zamiast BASIC'owej instrukcji PRINT czy PASCAL'owej WRITE język C wykorzystuje funkcje we/wy. Jest to cecha bardzo wygodna, ale oznacza że każda implementacja posiada swoją własną wersję podstawowych instrukcji we/wy. DEEP BLUE C nie stanowi tu wyjątku, ale jeżeli zaproponowany zbiór funkcji nie będzie Ci odpowiadał możesz zdefiniować swoje własne, bardziej odpowiadające Twoim potrzebom.

Funkcje zdefiniowane w plikach AIO.C, GRAPHICS.C, PMG.C i PRINTF.C zapewniają dostęp do możliwości sprzętowych komputera na takim poziomie jak BASIC. Funkcje biblioteczne o tak znanych nazwach jak *plot()*, *drawto()* czy *pokey()* działają podobnie jak ich odpowiedniki z języka BASIC. Pomijając oczywisty fakt, że najbardziej dokładną definicją każdej funkcji jest jej kod źródłowy w języku C podajemy jednak krótki opis każdej z nich.

A2. FUNKCJE WE/WY ZDEFINIOWANE W BIBLIOTECE AIO.C

```
clear( s, len)
char *s;
int len;
```

Funkcja clear ładuje do tablicy *s[0..len-1]* bajty zerowe, co jest wygodne przy inicjowaniu dużych tablic. Dla tablic złożonych z liczb całkowitych argument długości len powinien być pomnożony przez dwa, gdyż podaje się go w bajtach a nie w słowach.

```
copen( fn, mode)
char *fn, mode;
```

Funkcja copen otwiera plik, o nazwie zawartej w łańcuchu *fn*, dla odczytu, zapisu lub dołączania, w zależności od wartości znaku mode:

r - odczyt pliku (jak OPEN #n, 4, fn$)

w - zapis pliku (jak OPEN #n, 8, fn$)

a - dołączanie do pliku (jak OPEN #n, 12, 0, fn$)

Jeżeli plik zostanie otwarty poprawnie, to funkcja zwraca numer IOCB (0-7). Należy go zapamiętać w jakiejś zmiennej, gdyż będzie potrzebny do komunikowania się z danym plikiem. Jeżeli plik nie został otwarty funkcja zwraca liczbę ujemną, będącą negacją kodu błędu CIO. Na przykład wciśnięcie klucza BREAK w czasie próby otwarcia pliku spowoduje, że funkcja *copen()* zwróci wartość -128.

```
open( iocb, ax1, ax2, fname)
char iocb, ax1, ax2, *fname;
```

Funkcja ta jest podobna do instrukcji OPEN z języka BASIC.

Funkcja *open()* zwraca wartość 1 jeżeli nie ma problemów z otwarciem pliku. W przypadku przeciwnym zwraca ujemny kod błędu CIO.

```
close( i)
char i;
```

Ta funkcja jest podobna do instrukcji CLOSE z BASIC'a. Zamyka IOCB i zwraca wartość 1 lub w przypadku wystąpienia błędu ujemny kod błędu CIO.

```
cclose( i)
int i;
```

Gdy chcesz zamknąć konkretny plik powinieneś wywołać funkcję *cclose()* z liczbą zwracaną przez *copen()*. Funkcja cclose() zwraca wartość 1 gdy wszystko zakończyło się poprawnie lub negację błędu CIO, gdy nie udało się zamknąć pliku.

```
cgetc( iocb)
int iocb;
```

Funkcja *cgetc()* jest bardzo podobna do instrukcji GET z BASIC'a. Po przekazaniu numeru IOCB funkcja zwraca kolejny znak z pliku (o kodzie od 0 do 255) lub liczbę ujemną, która jest kodem błędu CIO.

```
getchar()
```

Funkcja *getchar()* wprowadza jeden znak z ekranu (iocb 0) i zwraca wartość 1 lub ujemny kod błędu CIO.

```
putchar( c)
char c;
```

Funkcja *putchar()* wyświetla dany znak na ekranie (iocb 0) i zwraca wartość 1 lub ujemny kod błędu CIO.

```
gets( string)
char *string;
```

Funkcja *gets()* jest podobna do instrukcji INPUT z BASIC'a. Wprowadza cały logiczny wiersz tekstu i umieszcza go w tablicy znakowej podanej jako argument. Tablica ta powinna mieć przynajmniej 120 bajtów długości, gdyż w przeciwnym przypadku operator może ją przepełnić poprzez wprowadzenie zbyt długiego wiersza. Jeżeli nie wystąpił błąd funkcja *gets()* zwraca liczbę znaków wprowadzonego wiersza (od 0 do 120). W przypadku wystąpienia błędu zwracany jest ujemny kod błędu CIO.

```
cprints( string)
char *string;
```

Funkcja *cprints()* jest podobna do instrukcji PRINT z BASIC'a. Wyświetla na ekranie łańcuch będący jej argumentem. Nie wyprowadza znaku RETURN ale do tego celu można wykorzystać instrukcję *putchar(155)*, która wymusi przejście kursora na początek następnego wiersza.

```
cputs( strig, iocb)
char *string;
int iocb;
```

Funkcja *cputs()* jest podobna do instrukcji PRINT # z BASIC'a. Wyprowadza przekazany łańcuch do konkretnego pliku. Należy jako drugiego argumentu funkcji użyć numeru IOCB zwróconego przez instrukcję *copen()*. Jeżeli brak błędu funkcja *cputs()* zwraca wartość 1, a w przypadku przeciwnym ujemny kod błędu CIO.

```
ciov( iocb, com, bad, blen, ax1, ax2)
int iocb, com, blen, ax1, ax2;
char *bad;
```

Funkcja *ciov()* jest podobna do wywołania XIO z BASIC'a. Możesz ustawić iocb i wywoływać przy pomocy tej funkcji CIO. Argument iocb powinien być z przedziału 0-7 i musi określać blok we/wy, którego używasz. com jest kodem rozkazu ICCOM, *bad* jest adresem bufora ICBAD, a *blen* długością bufora ICBLEN. *ax1* jest pomocniczym bajtem ICAX1, a *ax2* pomocniczym bajtem ICAX2. Jeżeli nie chcesz zmieniać bieżącej wartości któregoś z czterech ostatnich argumentów (*bad, blen, ax1, ax2*) użyj zastępczo wartości -1. Tak więc chcąc wczytać kolejny wiersz do bufora powinieneś napisać:

```
ciov(7, 5, -1, -1, -1, -1);
```

Większość podanych funkcji we/wy została zaimplementowana z wykorzystaniem *ciov()*. Dwa wyjątki *cgetc()* i *cputc()* zostały zaimplementowane w asemblerze, aby zwiększyć szybkość ich działania. Jeżeli CIO zwraca wynik mniejszy niż 128, to funkcja *ciov()* przekazuje go w niezmienionej postaci, ale jeżeli kod wynikowy jest większy lub równy 128 (oznacza to wystąpienie błędu), to *ciov()* zwraca ujemną wartość tego kodu. Takie podejście jest zgodne ze standardem języka C, w którym przyjęto, że kody błędów są podawane jako liczby ujemne.

```
normalize( fname, fext)
char *fname, *fext;
```

normalize() jest funkcją użytkową służącą do przekształcania dowolnej formy nazwy pliku w standardową postać wymaganą przez CIO i FMS. Najpierw nazwa pliku zamieniana jest na nazwę pisaną dużymi literami, a następnie, jeżeli nie występuje przedrostek nazwy urządzenia, na początek nazwy dodawane są znaki "D1:". Jeżeli brak rozszerzenia, to do nazwy pliku dodawana jest

kropka i łańcuch *fext*. Oto typowy przykład wykorzystania tej funkcji:

```
char fname[20];
gets( fname);
normalize( fname, "BAS");
```

Wykorzystanie tej funkcji gwarantuje otrzymanie nazwy pliku akceptowalnej przez system CIO. Na przykład jeżeli użytkownik wprowadzi nazwę "prog", to po

```
normalize( fname, "BAS");
```

łańcuch wynikowy zawiera "D:PROG.BAS"

```
toupper( c)
char c;
```

Jeżeli c jest małą literą, to funkcja *toupper()* zwraca kod odpowiedniej dużej litery.

```
tolower( c)
char c;
```

Jeżeli c jest dużą literą, to tolower() zwraca kod odpowiedniej małej litery.

```
strcpy( a, b)
char *a, *b;
```

Funkcja strcpy() kopiuje łańcuch znaków z tablicy b do tablicy a. Funkcja zwraca długość kopiowanego łańcucha z pominięciem bajtów zerowych.

```
move( a, b, len)
char *a, *b;
int len;
```

Funkcja move() przepisuje len bajtów z a do b zaczynając od elementu *a[0]*, a kończąc na *a[len-1]*. Dzieją się dziwne rzeczy jeżeli a <= b <= a + len.

```
usr( adr,...)
int adr;
```

Funkcja usr() jest podobna do funkcji USR() z BASIC'a. Pierwszy argument jest adresem podprogramu napisanego w języku maszynowym, natomiast reszta stanowi parametry jakie są do niego przekazywane. Wynik obliczeń zwracany jest w rejestrach A (część mniej znacząca) i X (część bardziej znacząca). Po wywołaniu podprogramu użytkownika stos wygląda następująco:

```
<oprócz adresu liczba argumentów (0-120)>
<bardziej znaczący bajt pierwszego argumentu>
<mniej znaczący bajt pierwszego argumentu>
<bardziej znaczący bajt drugiego argumentu>
itd.
<adres powrotu (dwa bajty)>
```

Procedura użytkownika może wykorzystywać bajty $f6-$ff strony zerowej.

```
find( addr, len, ch)
char *addr, ch;
int len;
```

Funkcja *find()* przeszukuje obszar pamięci od adresu *addr* do adresu *addr + len - 1*, aż do momentu wystąpienia znaku *ch*. Jeżeli funkcja nie znajdzie znaku ch zwraca wartość -1, a w przeciwnym przypadku podaje odległość (w bajtach) znaku ch od adresu *addr*.

```
peek( i)
char *i, d;
```

Funkcja *peek()* zwraca bajt pamięci o adresie i.

```
poke( i, d)
char *i, d;
```

Funkcja *poke()* wpisuje bajt d pod adres i i zwraca poprzednią zawartość tej komórki.

```
dpeek( i)
char *i;
```

Funkcja *dpeek()* zwraca zawartość słowa (dwa bajty) o adresie i.

```
dpoke( i, w)
char *i;
int w;
```

Funkcja *dpoke()* wstawia słowo w pod adres i, i+1 i zwraca poprzednią zawartość tego słowa.

```
val( s)
char *s;
```

Funkcja val(), podobnie jak instrukcja VAL z BASIC'a, zwraca wartość liczbową łańcucha s.

```
hval( s)
char *s;
```

Funkcja hval() zwraca wartość liczbową łańcucha s w kodzie szesnastkowym.

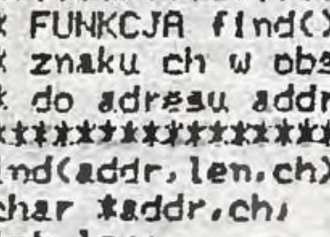
60 pp65 of6 lp54

POSTAC ZRODLOWA FUNKCJI ZAWARTYCH W ZBIORZE AIO.C

```
#define EOL 155
#define TAB 127
#define SPACE 32
/*****************************************************
 * FUNKCJA val(s) - zwraca do miejsca
 *  wywolania liczbe odpowiadajaca lancuchowi
 * (ciagowi cyfr) zawartemu w s
 *****************************************************
val(s)
 char *s;
{
 char c;
 int v,sign;
 while(c= *s){
  if(c==SPACE | c==EOL | c==TAB)++s;
  else break;
 }
 sign=1;
 if(c=='+' | c=='-'){
  if(c=='-')sign= -1;
  c= *++s;
 }
 v=0;
 while(c>='0' & c<='9'){
  v=v*10+c-'0';
  c=*++s;
 }
 return v*sign;
}
/*****************************************************
 * FUNKCJA hval(s)
 * zamienia liczbe szesnastkowa na
 * liczbe typu integer
 *****************************************************
hval(s)
 char *s;
 {
 int v;
 char c;
 v=0;
 while(c= *s){
  if(c==TAB | c==SPACE | c==EOL)++s;
  else break;
 }
 while(1){
  if(c>='0' & c<='9')c=c-'0';
  else if(c>='a' & c<='f')c=c-87;
  else if(c>='A' & c<='F')c=c-55;
  else break;
  v=(v<<4)+c;

  c=*++s;
 }
 return v;
}
/*****************************************************
 * FUNKCJA copen()
 * otwiera zbior o nazwie zawartej
 * w lancuchu fn
 *****************************************************
copen(fn,mode)
 char *fn,mode;
{
 int k,io,r,dir;
 io=0;
 while(io<8){
  if(peek(832+(io<<4))==255)break;
  ++io;
 }
 if(io==8)return -1; /* -1 otwarcie niemozliwe
 fn[k=strlen(fn)]=EOL;
 if(mode=='r')dir=4;
 else if(mode=='w')dir=8;
 else if(mode=='a')dir=9;
 else return -132;
 r=ciov(io,3,fn,k+1,dir,0);
 fn[k]=0;
 if(r<0){
  cclose(io);
```

```
  return r;      /* ujemny kod bledu */
 $)
 return iocb;    /* numer iocb       */
$)
/*****************************************************
 * FUNKCJA open()
 * otwiera blok IOCB do operacji we/wy
 *****************************************************/
open(iocb,ax1,ax2,fname)
 char iocb,ax1,ax2,*fname;
$(
 return ciov(iocb,3,fname,
  strlen(fname),ax1,ax2);
$)
/*****************************************************
 * FUNKCJA close(i)
 * zamyka blok IOCB
 *****************************************************/
close(i)
 char i;
$(
 return cclose(i);
$)
cclose(i)
 int i;
$(

 return ciov(i,12,-1,-1,-1,-1);
$)
/*****************************************************
 * FUNKCJA coutc() - przesyla znak
 * do urzadzenia skojarzonego
 * z otwartym blokiem IOCB
 *****************************************************/
coutc(c,unit)
 char c;
 int unit;
asm 12297;
/*****************************************************
 * FUNKCJA cgetc()
 * odczytuje kolejny znak z urzadzenia
 * skojarzonego z otwartym blokiem IOCB
 *****************************************************/
cgetc(unit)
 int unit;
 asm 12294;
/*****************************************************
 * FUNKCJA getchar()
 * wczytuje jeden znak z ekranu (IOCB=0)
 *****************************************************/
getchar()
$(
 return cgetc(0);
$)
/*****************************************************
 * FUNKCJA putchar()
 * przesyla jeden znak na ekran (IOCB=0)
 *****************************************************/
putchar(c)
 char c;
$(
 return coutc(c,0);
$)
/*****************************************************
 * FUNKCJA gets(str)
 * wprowadza logiczny wiersz tekstu
 * z ekranu (IOCB=0) do lancucha str
 *****************************************************/
gets(str)
 char *str;    /* wskaznik do lancucha str */
$(
 int r;
 if((r=ciov(0,5,str,120,-1,-1))<0)
  return r;
 str[r=dpeek(840)-1]=0;
 return r;
$)
/*****************************************************
 * FUNKCJA cprints()
 * wyswietla na ekranie lancuch znakow
 * zawarty w lancuchu str

 *****************************************************/
cprints(str)
 char *str;
$(
 return couts(str,0);
$)
/*****************************************************
 * FUNKCJA couts() - wysyla na urzadzenie
 * skojarzone z otwartym blokiem IOCB
 * lancuch znakow zawarty w str
 *****************************************************/
couts(str,i)
 char *str;
 int i;
$(
 int k;
 if(k=strlen(str))
  return ciov(i,11,str,k,-1,-1);
 else return i;
$)
/*****************************************************
 * FUNKCJA ciov() - gdy argumenty bad, bien, ax1
 * lub ax2 sa przy wywolaniu ustawione na -1,
 * to ich biezace wartosc nie ulega zmianie
 *****************************************************/
ciov(iocb,com,bad,bien,ax1,ax2)
 int iocb,com,bien,ax1,ax2;
 char *bad;
asm 12291; /* $3003 */
/*****************************************************
 * FUNKCJA normalize()
 * zamienia litery z nazwy zbioru na duze litery
 * dodaje przedrostek D:, gdy go brak
 * dodaje rozszerzenie zdefiniowane przez fext
 *****************************************************/
normalize(fname,fext)
 char *fname,*fext; /* wskazniki do nazwy i rozsz.
$(
 int i;
 char c,temp[20];
 /* zamiana nazwy zbioru na duze litery */
 i=0;
 while(c=fname[i])
   fname[i++]=toupper(c);
 if(--i < 0)return 0; /* nazwa pusta (brak) */
 /* fname[i] jest ostatnim znakiem nazwy */
 /* przedrostek D: */
 if(i==0 | (i==1 & fname[1]!=':')
  |(fname[1]!=':'&fname[2]!=':'))$(
  strcpy(temp,"D:");
  strcpy(temp+2,fname);
  strcpy(fname,temp);
  i=i+2;
 $)


 /* gdy brak '.', dodaj '.'fext */
 if(find(fname,i+1,'.')<0)$(
  fname[i+1]='.';
  strcpy(fname+i+2,fext);
 $)
$)
/*****************************************************
 * FUNKCJA toupper() - zwraca kod duzej
 * (A .. Z), gdy argumentem
 * jest kod malej litery (a .. z)
 *****************************************************/
toupper(c)
 char c;
$(
 return (c < 'a') || (c > 'z')
  ? c : c-32;
$)
/*****************************************************
 * FUNKCJA tolower() - zwraca kod malej
 * (a .. z), gdy argumentem
 * jest kod duzej litery (A .. Z)
 *****************************************************/
tolower(c)
 char c;
$(
 return (c < 'A') || (c > 'Z')
  ? c : c+32;
$)
/*****************************************************
 * FUNKCJA strcpy(a,b)
 * kopiuje lancuch "b" do "a"
 *****************************************************/
strcpy(a,b)
 char *a,*b; /* wskazniki do kop. lancuchow */
asm 12300;
/*****************************************************
 * FUNKCJA move(a,b,len)
 * przepisuje len znakow z "a" do "b"
 *****************************************************/
move(a,b,len)
 char *a,*b;
 int len;
asm 12303;
/*****************************************************
 * FUNKCJA clear(a,len) -- zeruje len elementow
 * tablicy "a" poczawszy od a[0] */
```

```
clear(a,len)
 char *a;
 int len;
${
 *a=0;
 move(a,a+1,len-1);
$}
/*************************************************

 * FUNKCJA usr() - wywolanie podpr. maszynowego
 * rezydujacego od adresu addr
 *************************************************
usr(addr)
 char *addr;
asm 12306;
/*************************************************
 * FUNKCJA find() - szuka pierwszego wystap.
 * znaku ch w obszarze pamieci od adresu addr
 * do adresu addr+len-1
 *************************************************
find(addr,len,ch)
 char *addr,ch;
 int len;
asm 12309;
/*************************************************
 * FUNKCJA strlen(str) - zwraca liczbe
 * znakow zawartych w lancuchu str
 *************************************************
strlen(str)
 char *str;
${
 return find(str,30000,0);
$}
/*************************************************
 * FUNKCJA peek(i) - zwraca zawartosc
 * bajtu o adresie i
 *************************************************
peek(i)
 char *i;
asm 12312;
/*************************************************
 * FUNKCJA poke(i,d) - wpisuje wartosc
 * d do bajtu o adresie i i zwraca
 * stara zawartosc tego bajtu */
poke(i,d)
 char *i,d;
asm 12315;
/*************************************************
 * FUNKCJA dpeek(i) - zwraca zawartosc slowa
 * (dwoch bajtow) o adresie i
 *************************************************
dpeek(i)
 char *i;
asm 12318;
/*************************************************
 * FUNKCJA dpoke(i,w)
 *************************************************
dpoke(i,w)
 char *i;
 int w;
asm 12321;
```



A3. FUNKCJE GRAFICZNE ZDEFINIOWANE W BIBLIOTECE GRAPHICS.C

```
graphics( n)
char n;
```

Funkcja *graphics()* zmienia tryb graficzny ekranu podobnie jak instrukcja GRAPHICS w języku BASIC. Zwraca takie same wartości jak funkcja *open()*.

```
color( c)
char c;
```

Funkcja *color()* ustawia kolor pisaka dla funkcji *plot()* lub *drawto()* podobnie jak instrukcja COLOR w BASIC'u. Zwraca wartości przypadkowe.

```
drawto( x, y)
int x, y;
```

Funkcja *drawto()*, podobnie jak instrukcja DRAWTO w BASIC'u, kreśli linię od ostatnio oznaczonego punktu (*plot()*) do punktu o współrzędnych (x, y). Zwraca wartość 1 gdy nie ma błędów lub ujemny kod błędu CIO.

```
locate( x, y)
int x, y;
```

Funkcja *locate()* umieszcza kursor graficzny w pozycji (x,y) i zwraca wartość tego piksla lub ujemny kod błędu CIO. Stanowi ona odpowiednik instrukcji LOCATE w języku BASIC.

```
plot( x, y)
int x, y;
```

Funkcja *plot()* zapala punkt (x,Y) tak jak instrukcja PLOT języka BASIC. Zwraca wartość 1 lub kod błędu CIO.

```
position( x, y)
int x, y;
```

Funkcja *position()* umieszcza kursor graficzny w nowym miejscu o współrzędnych (x,y). Kursor zmienia swoje położenie dopiero przy kolejnej instrukcji wyprowadzania.

```
setcolor( reg, hue, lum)
char reg, hue, lum;
```

Funkcja *setcolor()* ustawia kolor rejestru reg kombinacja wartości hue i lum. Działa podobnie jak instrukcja SETCOLOR BASIC'a.

```
fill( x, y, c)
int x, y;
char c;
```

Funkcja *fill()* stanowi implementacje rozkazu FILL BASIC'a dla urządzenia 8 (ekran). Kreśli linię od ostatnio narysowanego punktu do punktu o współrzędnych (x,y) zmieniając kolor tła po prawej stronie kreślonej linii. Jest bardzo przydatna przy wypełnianiu kolorem dużych trapezoidalnych obszarów ekranu.

```
paddle( n)
char n;
```

Funkcja *paddle()* zwraca wartość wiosełka określonego przez n. Działa tak jak instrukcja PADDLE języka BASIC.

```
ptrig( n)
char n;
```

Funkcja ptrig() zwraca wartość przycisku określonego wiosełka. Działa podobnie jak instrukcja PTRIG BASIC'a.

```
stick( n)
char n;
```

Funkcja *stick()* zwraca wartość określonego dżojstika. Działa podobnie jak instrukcja STICK w języku BASIC.

```
strig( n)
char n;
```

Funkcja *strig()* zwraca wartość przycisku określonego dżojstika. Działa podobnie jak funkcja STRIG BASIC'a.

```
vstick( n)
char n;
```

Funkcja *vstick()* zwraca pionową składową położenia dżojstika. Gdy jest wychylony w górę zwraca wartość 1, gdy w dół -1, natomiast gdy znajduje się w położeniu neutralnym wartość 0.

```
hstick( n)
char n;
```

Funkcja *hstick()* zwraca poziomą składową położenia dżojstika. Gdy jest wychylony w lewo zwraca wartość -1, gdy w prawo 1, a gdy jest w położeniu neutralnym 0.

```
sound( voice, pitch, dist, volume)
char voice, pitch, dist, volume;
```

Funkcja *sound()* wywołuje efekty dźwiękowe podobnie jak instrukcja SOUND BASIC'a.

```
rnd( n)
int n;
```

Funkcja *rnd()* zwraca liczbę losową z zakresu od 0 do n-1 włącznie. Jeżeli chcemy uzyskać liczbę losową z zakresu od i do 10 należy użyć wyrażenia i+rnd(10). Jeżeli n jest mniejsze niż 2, to funkcja *rnd()* zwraca wartość 0.

A4. FUNKCJE GRAFIKI PMG ZDEFINIOWANE W BIBLIOTECE PMG.C

```
pmcinit()
```

Funkcja *pmcinit()* inicjuje grafikę "graczy i pocisków" oraz zbiór znaków. Funkcja *pmcinit()* powinna być użyta tylko raz, przed wywołaniem funkcji *graphics()*.

```
pmcflush()
```

Funkcja *pmcflush()* kasuje bufory znaków graficznych i grafiki PMG zwalniając 4K bajtów RAM. Funkcja *pmcflush()* powinna być użyta tylko raz, tuż przed powrotem do systemu operacyjnego.

```
pmgraphics( i)
int i;
```

Funkcja *pmgraphics()* powinna być wywoływana po każdym odwołaniu do *graphics()*, aby ustawić rozdzielczość grafiki PMG. Funkcja *pmgraphics(1)* ustawia rozdzielczość jednoliniową, *pmgraphics(2)* ustawia rozdzielczość dwuliniową, a *pmgraphics(0)* wstrzymuje grafikę PMG.

```
hitclear()
```

Funkcja *hitclear()* czyści rejestry kolizji.

```
hitp2pf( from, to)
char from, to;
```

Funkcja *hitp2pf()* zwraca wartość 1 gdy gracz *#from* uderzy w pole *#to*.

```
hitp2pl( from, to)
char from, to;
```

Funkcja *hitp2pl()* zwraca wartość 1, gdy gracz *#from* trafi w gracza *#to*, a 0 gdy nie trafi. Gdy *#from* jest równe *#to*, to funkcja zwraca wartość 1.

```
pmclear( n)
char n;
```

Funkcja *pmclear()* zmazuje gracza n.

```
pmcolor( n, c, i)
char n, c, i;
```

Funkcja *pmcolor()* ustawia kolor gracza/pocisku na c, a jasność na i.

```
pmwidth( n, w)
char n, w;
```

Funkcja *pmwidth()* ustawia szerokość gracza n na wartość w:

w==0 normalny wymiar;
w==1 podwójny wymiar;
w==2 poczwórny wymiar;

```
pladdr( n)
char n;
```

Funkcja *pladdr()* zwraca adres bufora zawierającego gracza n.

```
plmove( n, x, y, shape)
char n, x, y, *shape;
```

Funkcja *plmove()* przesuwa gracza n na pozycję (x,y) (we współrzędnych aktualnego trybu graficznego) i pobiera jego kształt z tablicy znakowej *shape*. *Shape[0]* to wielkość gracza, a *shape[1..size]* to bajty tworzące kształt gracza. Należy zostawić kilka bajtów zerowych przed i po, aby mieć pewność, że poprzednie kształty zostaną prawidłowo wymazane.

```
chget( c, s)
char c, *s;
```

Funkcja *chget()* wypełnia tablicę *s[0..7]* zawierającą kształt wewnętrznego znaku Atari, którego kod znajduje się w c.

```
choget( c, s)
char c, *s;
```

Funkcja *choget()* wypełnia tablicę *s[0..7]* znakiem oryginalnym z ROM.

```
chput( c, s)
char c, *s;
```

Funkcja *chput()* wypełnia zawartość tablicy *s[0..7]* dla znaku c. Aby umieścić np. kropkę na środku pustego pola znaku należy napisać:

```
chput(0,"\0\0\0\60\0\0\0");
```

UWAGA: Przed wykorzystaniem jakiejkolwiek funkcji z biblioteki PMG.C należy wywołać najpierw funkcję *pmcinit()*.

A5. FUNKCJE ZDEFINIOWANE W PLIKU PRINTF.C

Oprócz funkcji znajdujących się w pliku AIO.C istnieją w pliku PRINTF.C jeszcze dwie funkcje biblioteczne.

```
printf(s,...)
char *s;
```

Funkcja *printf()* jest standardową funkcją sformatowanego wyprowadzania zaimplementowaną w języku C. Akceptuje ona zmienną liczbę argumentów. Pierwszym z nich jest łańcuch formatujący, zawierający tekst jaki ma być wyprowadzony i znaki określające gdzie wstawić resztę argumentów. Znak % występujący w łańcuchu formatującym ma znaczenie specjalne. Znaki występujące po % określają sposób drukowania kolejnego argumentu. Pierwszy znak % dotyczy pierwszego argumentu, drugi znak % drugiego argumentu, itd. Jeżeli podano zbyt mało argumentów (lub zbyt mało znaków %), to zostaną wyprowadzone przypadkowe wartości.

Po znaku % można napisać jedną z następujących liter:

d - aby wydrukować liczbę dziesiętną
x - aby wydrukować liczbę szesnastkową
c - aby wydrukować znak
s - aby wydrukować łańcuch znaków
% - aby wydrukować znak %

Jeżeli chcesz aby argument miał określoną liczbę znaków podaj ją między znakiem % a znakiem formatu.

Standardowo następuje równanie drukowanej wartości do lewego krańca pola. Jeżeli chcesz aby równanie następowało do prawego skraju wstaw przed liczbą znak minus (-). Oto kilka przykładów:

```
printf("abcd");  -> abcd
printf("=%s=","abcd");  -> =abcd=
printf("=%3d=", 99);  -> =99 =
printf("=%-3d=",99);  -> =  99=
printf("%c %d %x", 65, 65, 65);  -> A 65 41
```

```
fprintf(iocb, s, ...)
int iocb;
```

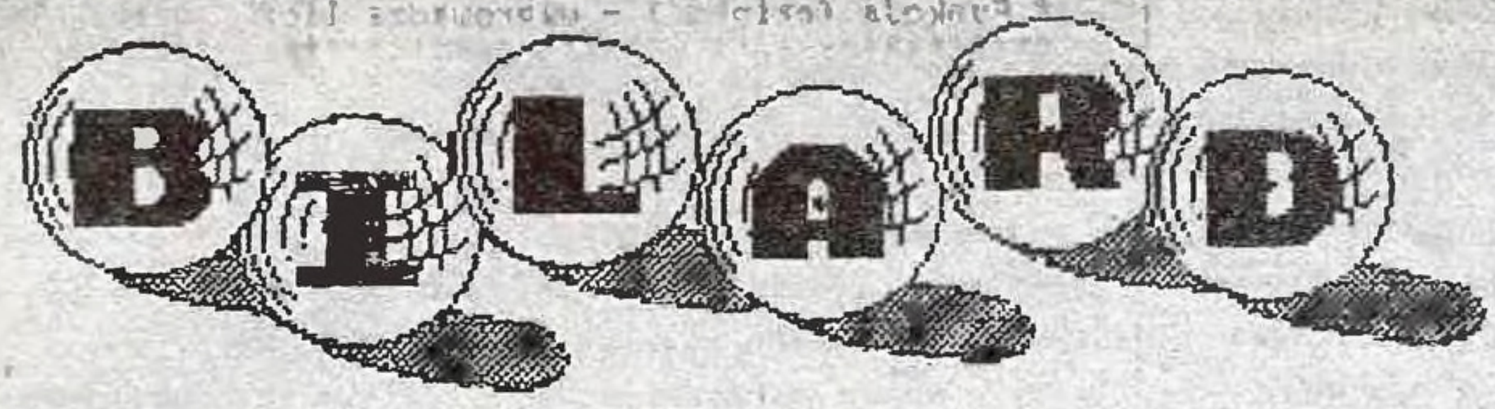
char *s)

Funkcja *fprintf()* jest podobna do *printf()*, ale ma dodatkowy argument *iocb* i wyprowadza dane do pliku o tym *iocb*. Funkcja *printf()* to właściwie *fprintf(0,...)*.

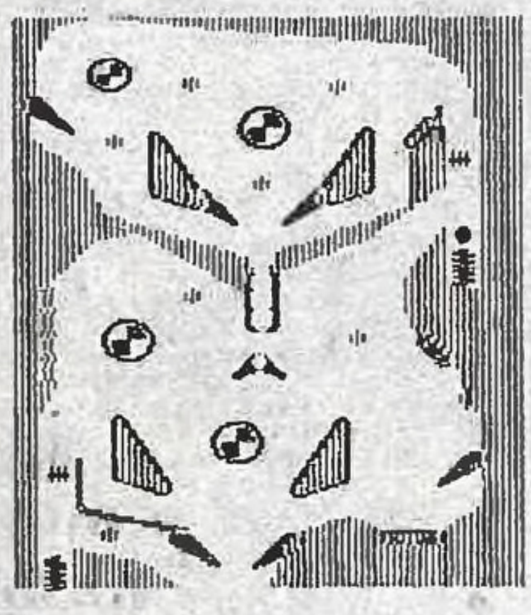
POSTAC ZRODLOWA FUNKCJI ZAWARTYCH
W ZBIORZE PRINTF.C

```
/* funkcje wyprowadzania sformatowanego */

/*****************************************************
 * FUNKCJA printf()
 *****************************************************
printf()
${
 int sp;
 sp=dpeek(210);
 sp=sp-(6+peek(dpeek(sp-8)));
 return fprint(sp,0);
$}
/*****************************************************
 * FUNKCJA fprintf()
 *****************************************************
fprintf()
${
 int sp;
 sp=dpeek(210);
 sp=sp-(6+peek(dpeek(sp-8)));
 return fprint(sp+2,dpeek(sp));
$}
fprint(s,io)
 char *s;
 int io;
${
 char *fstring,*arg,c;
 int cleft,i,size,sign;
 cleft=strlen(fstring=dpeek(s));
 while(cleft)${
  if((i=find(fstring,cleft,'%'))<0)${
   cputs(fstring,io);
   return; /* nie ma wiecej zn. formatujacych */
  $}
  if(i>0)${ /* znak przed % */
   ciov(io,11,fstring,i,-1,-1);
   fstring=fstring+i;
   cleft=cleft-i;
  $}
 /* pobierz argument numeryczny*/
 size=0;
 sign=1;
 while(cleft)${
  ++fstring;
  --cleft;
  c= *fstring;
  if(c=='-')sign= -sign;
  else if(c>='0' & c<='9')
   size=size*10+(c-'0');
  else break;
 $}
 size=size*sign;

 arg=dpeek(s=s+2);
 /* sprawdzenie znaku formatujacego */
 /* tj. znaku wystepujacego po %    */
 if(c=='d')fprintd(arg,size,io);
   /* gdy znak formatujacy = d,     */
   /* to funkcja fprintd            */
 else if(c=='s')
   /* gdy znak formatujacy = s,     */
   /* to funkcja fprints            */
  fprints(arg,size,io);
 else if(c=='c')
   /* gdy znak formatujacy = c,     */
   /* to funkcja fprintc            */
  fprintc(arg,size,io);
 else if(c=='x')
   /* gdy znak formatujacy = x,     */
   /* to funkcja fprintx            */
  fprintx(arg,size,io);
 else ${ /*  %%, etc. */
   /* gdy znak formatujacy = %,     */
   /* to wyprowadz ten znak         */
  cputc(c,io);
  s=s-2; /* i w tej sytuacji brak   */
         /* argumentu do wyprow.    */
 $}
 ++fstring;
 --cleft;
$}
/*****************************************************
```

```
 * Funkcja fprintd() - wyprowadza liczbe dzies.
 *****************************************************/
fprintd(arg,size,io)
 int arg,size,io;
${
 /* bufor na lancuch odpowiadajacy   */
 /* wyprowadzanej liczbie            */
 char buf[7],flag,c;
 int i,k;
 i=flag=0;
 k=10000;
 if(arg<0)${
  buf[i++]='-';
  arg= -arg;
 $}
 while(k)${
  if(flag | (c=arg/k)>0)${
   buf[i++]=c+'0';
   flag=1;
  $}
  arg=arg%k;
  k=k/10;
 $}
 if(flag==0) /* zero */

  buf[i++]='0';
 buf[i]=0;
 if(size > 0)
  fprintws(io,size-i);
 cputs(buf,io);
 if(size < 0)
  fprintws(io,(-size)-i);
$}
/*****************************************************
 * FUNKCJA fprintx() - wyprowadza liczbe hex.
 *****************************************************
fprintx(arg,size,io)
 int arg,size,io;
${
 char buf[7],flag,c;
 int i,field;
 i=flag=0;
 field=12;
 while(field>=0)${
  if(flag |
   (c=(arg >> field)&15)>0)${
   if(c<=9)buf[i++]=c+'0';
   else buf[i++]=c+'a'-10;
   flag=1;
  $}
  field=field-4;
 $}
 if(flag==0) /* zero */
  buf[i++]='0';
 buf[i]=0;
 if(size>0)fprintws(io,size-i);
 cputs(buf,io);
 if(size<0)fprintws(io,(-size)-i);
$}
/*****************************************************
 * FUNKCJA fprintc() - wyprowadza znak
 *****************************************************
fprintc(arg,size,io)
 char arg;
 int size,io;
${
 if(size>0)fprintws(io,size-1);
 cputc(arg,io);
 if(size<0)fprintws(io,(-size)-1);
$}
/*****************************************************
 * FUNKCJA fprints() - wyprowadza lancuch znakow
 *****************************************************
fprints(arg,size,io)
 char *arg;
 int size,io;
${
 int i;
 i=strlen(arg);

 if(size>0)fprintws(io,size-i);
 cputs(arg,io);
 if(size<0)fprintws(io,(-size)-i);
$}
/*****************************************************
 * FUNKCJA fprintws() - wyprowadza n odstepow
 *****************************************************
fprintws(io,n)
 int io,n;
${
 while(n-- > 0)cputc(' ',io);
$}
```



Jednymi z pierwszych automatów do gry , instalowanych w kraju, były bilardy . Zasada gry między poszczególnymi jego rodzajami mimo różnego stopnia skomplikowania czy kształtu automatu była ta sama. Po wrzuceniu monety wybijało się kulkę , a następnie dwoma przyciskami sterowało zestawem dźwigni tak , aby kulka jak najdłużej odbijała się od tych dźwigni oraz innych elementów automatu . Każde takie odbicie wiązało się z uzyskaniem kolejnych punktów , premii w postaci punktów lub dodatkowej kulki . Zdenerwowany gracz niejednokrotnie , chcąc przechylić szale zwycięstwa na swoją korzyść , przyłożył pięścią w obudowę lub zaczął szarpać całym automatem . Wszystkie chwyty były dozwolone , byle tylko jak najdłużej utrzymać się przy grze . Powyższe metody zdobywania punktów stają się nieaktualne w przypadku przedstawionych poniżej bilardów komputerowych . Gier tego typu stworzono wiele. Różnica między nimi polega na wymyślnej grafice , bogactwie dźwięku , itd. W pierw-



Pinball 3

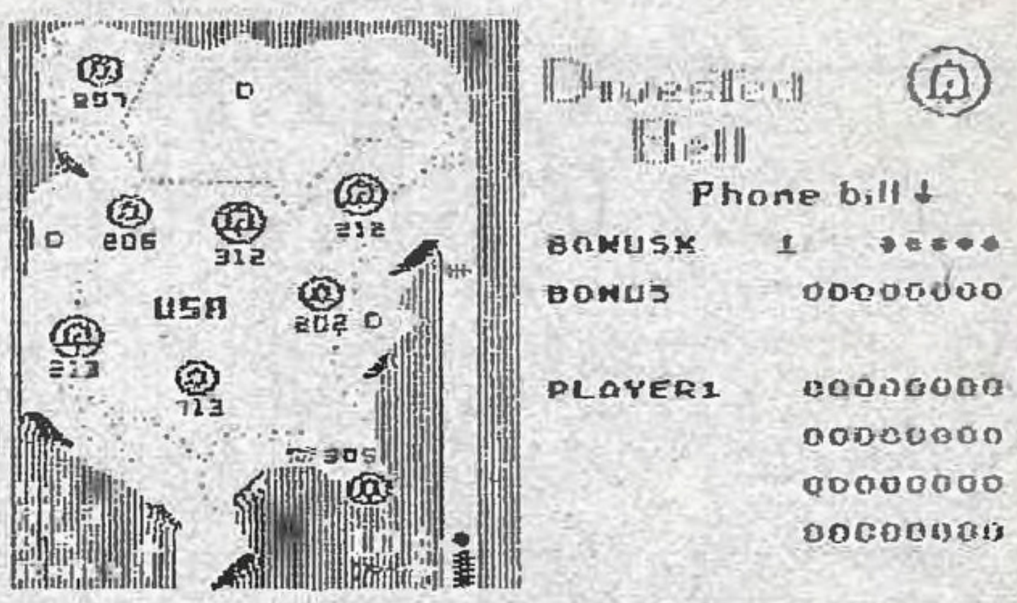
BONUSX	1	••••
BONUS		00003000
PLAYER1		01193710
		00000000
		00000000
		00000000

szych trzech przykładach , tzn. PINBALL 3 , MERRY CHRISTMAS i DIVESTED BELL obsługa jest identyczna. Punktacja może być prowadzona jednocześnie dla czterech graczy . W tym celu naciskamy klawisz **OPTION** tyle razy , ile osób bierze udział w grze . Aby kulka została wystrzelona , należy uprzednio spowodować naciągnięcie sprężyny klawiszem **START**

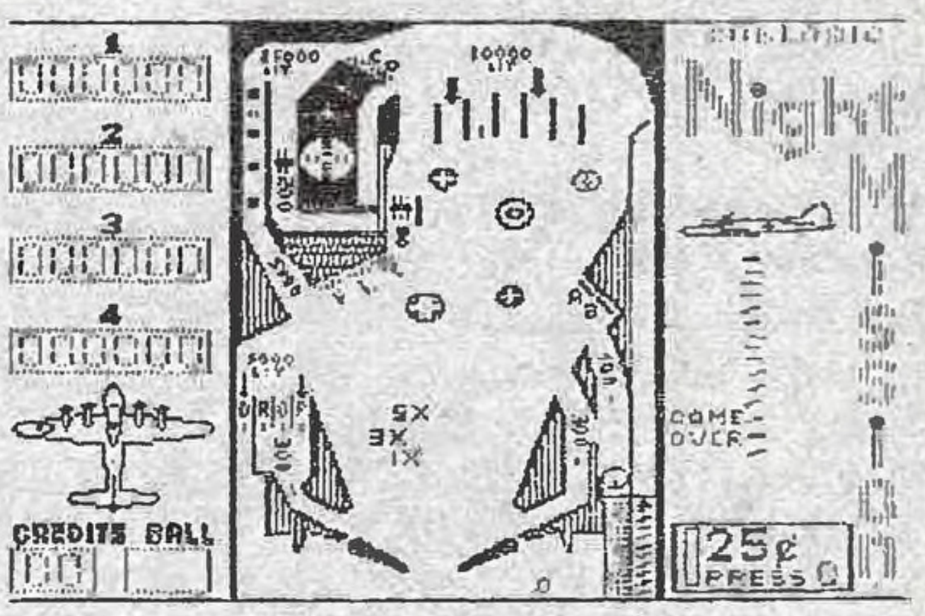
MERRY Christmas

BONUSX	2	•••
BONUS		00004000
PLAYER1		00074950
		00000000
		00000000
		00000000

lub przyciskiem FIRE manipulatora podłączonego do portu pierwszego. Ze wszystkich tych trzech dysponujesz pięcioma seriami rzutów. W każdej z tych gier przypada inna ilość kulek na serię. DIVESTED BELL - jedna , MERRY CHRISTMAS - dwie i PINBALL 3 od trzech wzwyż ; górna granica zależna jest od ilości zdobytych punktów , za które możesz otrzymać premię w postaci dodatkowych kulek . W grze możesz stosować jeden lub dwa manipulatory . Jeżeli masz jeden manipulator , używaj portu 1 . Użycie przycisku manipulatora powoduje wystrzelenie kulki . Ruch lewych dźwigni w bilardzie uzyskuje się przez naciskanie przycisku lub zwrot drążka manipulatora w lewo , natomiast ruch prawych dźwigni przez zwrot drążka w prawo . Mając dwa manipulatory , używa się tylko przycisków manipulatorów . Czasami zachodzi konieczność zwiększenia lub zmniejszenia naciągu sprężyny wystrzeliwującej kulkę . Efekt ten otrzymuje się , kierując drążek manipulatora do siebie lub od siebie .



Czwarty program NIGHT MISSION stanowi znacznie bogatszą ofertę niż poprzednia . Zarówno po stronie wizualnej jak i dźwiękowej wykorzystano w pełni możliwości Atari . Osobie stojącej w pobliżu mogłoby się wydawać , że to nie bilard a gra symulująca walkę powietrzną . Aby jednak mogło to mieć miejsce , należy wrzucić kilka monet 25 c , by otworzyć sobie kredyt . Robi się to klawiszem 0 . Ilość gier , za które zapłaciliśmy , możemy odczytać pod napisem **CREDITS**. Następnie klawiszem S zerujemy cztery tablice wyników po lewej stronie ekranu . Jedno naciśnięcie tego klawisza oznacza



że w grze uczestniczył będzie jeden gracz . Dalsze naciskanie umożliwi grę kolejnym chętnym (maks. czterem) . W tym czasie następuje również automatyczne naciągnięcie sprężyny wyrzucającej kulki . Naciąg tej sprężyny możesz regulować podobnie jak w poprzednich grach : manipulatorem podłączonym do portu 1 (do siebie naprężanie , od siebie zmniejszenie naciągu sprężyny) . Ruchem w lewo lub prawo tego manipulatora ustawia się kolor ramki . Wystrzelenie kulki następuje po naciśnięciu przycisku jednego z manipulatorów . Manipulatorem z portu 2 można wykonywać jeszcze dwie czynności : zmiana koloru planszy (drążek w lewo lub prawo) . zmiana nasycenia koloru (drążek w góre lub dół). Stwarza to możliwość wprowadzenia utrudnień . W celu poprawienia ci możliwości orientacji jak długo będziesz jeszcze grał , pod napisem BALL podawana jest liczba wystrzelonych kulek .

MOVIE MUSICAL MADNESS

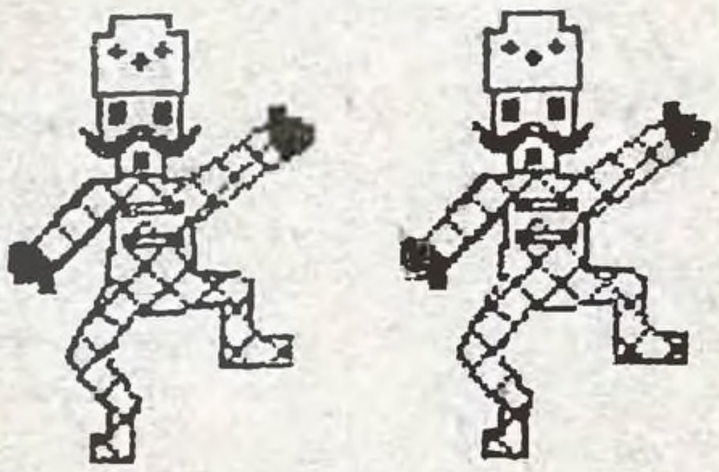
Jeżeli chcesz pobawić się w tworzeniu scenografii filmowej , a także komponowania do niej akcji , załaduj ten program . Gdy pojawią się na ekranie jego wstępne możliwości : COLOR (kolor) czy BLACK & WHITE (czarno - białe) ; ustaw manipulatorem gwiazdkę nad tym wariantem , który wybierasz , a następnie naciśnij przycisk . Pojawią się trzy postacie . Lewa z literką S na piersiach, prawa z literką C. Wybranie którejś z nich następuje przez przesunięcie manipulatorem w lewo lub prawo znajdującej się nad nimi gwiazdki , a następnie po naciśnięciu przycisku. Postacie znikają , pozostają tylko dwa rzędy po 20 prostokątów (góra i dół ekranu) . W tym momencie program jest gotów do tworzenia scenografii . Dolny rząd prostokątów to 20 różnych melodii , które możesz wykorzystać przy realizacji swojego dzieła. Górny rząd to twój magazyn rekwizytów , a pierwszy i ostatni prostokąt są "garderobami" statystów, którzy mogą uatrakcyjnić twoje dzieło . Niestety, w scenie może brać udział tylko jeden aktor i jeden statysta . Przy pomocy pierwszego prostokąta możesz otrzymać następujących statystów :

- czarodziej ;
- zbójnik ;
- statek kosmiczny ;
- pies ;
- reklama (ENT)

Dwudziesty prostokąt daje Ci

- statek kosmiczny ;
- babę Jagę ;
- pociąg ;
- kosmoludka ,
- duszka

Aby otrzymać np. zbójnika, należy manipulatorem nasunąć migającą gwiazdkę na pierwszy prostokąt , nacisnąć jego przycisk , ukazują - cego się czarodzieja skasować klawiszem SPACE i ponownie nacisnąć przycisk manipulatora, a gdy ukaże się zbójnik , potwierdzić jego wybór przez naciśnięcie przycisku .



Innych statystów uzyskuje się identycznie w kolejności, w jakiej zostały wyszczególnione powyżej . Teraz możesz przystąpić do tworzenia scenografii . Poszczególne jej elementy wybieramy identycznie jak statystów z tym , że jeden prostokąt odpowiada jednemu rekwizytowi. Jest ich w sumie osiemnaście . Możesz je ustawić dowolnie na planie, lecz pamiętaj , że dopóki nie potwierdzisz przez naciśnięcie przycisku manipulatora , możesz go dowolnie przemieszczać na planie lub skasować klawiszem spacji . A oto wybór wszystkich rekwizytów , jakie możesz otrzymać w kolejności od strony lewej :

1. baszta ;
2. minaret ;
3. dom ;
4. fragment muru ;
5. palma ;
6. drzewo ;
7. płot ;
8. pianino ;
9. planeta ;
10. tulipan ;
11. drogowskaz ;
12. dom (hotel) ;
13. wieżowiec ;
14. stół ;
15. reklama (SHOW) ;
16. księżyc ;
17. latarnia uliczna ;
18. stara lampa .

Sposób ich doboru oraz ilość jest zupełnie dowolna . Kiedy uznasz że scenografia jest już dostatecznie rozbudowana , przychodzi kolej na dobranie do niej odpowiedniego podkładu muzycznego . Przejście z górnego rzędu prostokątów następuje przez skierowanie drążka manipulatora do dołu i następnie wybraniu jednej z dwudziestu melodii . Przy jej wyborze postępuje się identycznie jak przy rekwizytach . Gdy dojdziesz do wniosku , że melodia odpowiada twojej koncepcji , naciśnij jeszcze raz przycisk manipulatora . Pojawi się polecenie PRESS START TO BEGIN ACTING . Naciśnij wtedy klawisz

START . Rozpocznie się realizacja sceny filmu . Przemieszczaj postać zgodnie z Twoją koncepcją . Tworzysz w ten sposób akcję danej sceny . Czas trwania sceny jest uzależniony od długości melodii . Jeżeli uprzednio był wprowadzony aktor np. czarodziej , pies itp. , pojawi się on i będzie wykonywał charakterystyczną dla siebie czyn-

ność . Koniec melodii oznacza koniec sceny . Na ekranie pojawiają się ponownie trzy postacie. Możesz rozpocząć tworzenie scenografii następnej sceny jak również jej pozostałych elementów , np. zmiana statystyk , melodii , itp. Możesz również poprzez naciśnięcie klawisza SELECT przejść do nowej opcji. tzn. RUSHES (odtwarzanie całego utworu) , RETAKES (zlikwidowanie ostatnio utworzonej sceny) , THE END (koniec filmu). Jak zwykle , wyboru dokonuje się przy pomocy manipulatora. Nie należy przerażać się przypadkowym wybraniem opcji THE END , ponieważ komputer prezentuje Ci cały film , a następnie proponuje: PRESS OPTION TO VIEW IT AGAIN (ponowne odtworzenie filmu po naciśnięciu klawisza OPTION) i PRESS SELECT TO MAKE A NEW FILM (skasowanie stworzonego filmu , a następnie przejście do tworzenia nowego przez naciśnięcie klawisza SELECT) . Przedstawiony powyżej program jak również jego możliwości z pewnością nie usatysfakcjonują wszystkich . Brak jest tu działań laserowych , strzelających kosmitów czy zdradliwych tuneli. Jednak szansa sprawdzenia się w bądź co bądź prestiżowym zawodzie powinna skusić wielu młodych ludzi . Nikt nie rodzi się reżyserem czy scenografem , tego trzeba się nauczyć .

HACKER

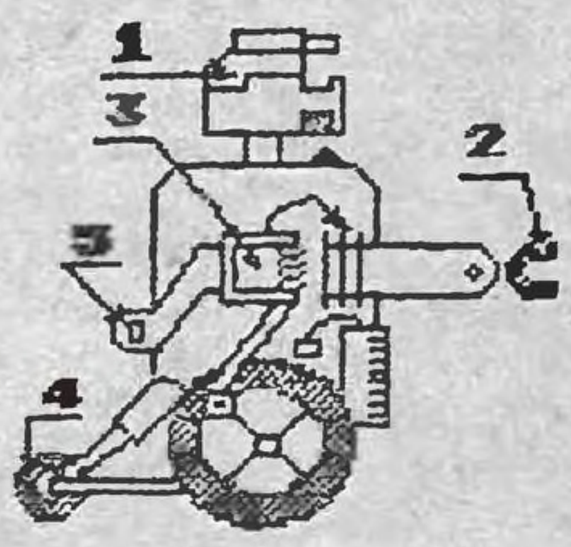
Właściwie nie wiadomo, w jakim celu i kiedy została wybudowana sieć tuneli łącząca prawie wszystkie stolice i największe miasta świata. Do naszych czasów dotrwały w stanie nienaruszonym tylko ich fragmenty. Dzięki przezorności Narodów Zjednoczonych zostały wyremontowane , a następnie zaadoptowane do super szybkiego przemieszczania ludzi i sprzętu . I nadszedł taki czas , kiedy trzeba było praktycznie to wykorzystać . Wydzielona grupa przedstawicieli Narodów Zjednoczonych opracowała plan ochrony Ziemi przed intruzami z zewnątrz . Nazwano go planem MAGMA . Aby nie dopuścić do rozszyfrowania go w razie wykradzenia , został on podzielony na dziewięć części.Niestety , jak można było przypuszczać, wrogowie dowiedzieli się o istnieniu planu . Polecili swoim szpiegom wykraść go . Szpiedzy dokonali tego . W obecnej chwili poszczególne części planu znajdują się w różnych miastach rozrzuconych po całej kuli ziemskiej . Jeżeli w odpowiednim czasie nie odzyskasz skradzionych części planu, wrogowie przejmą kontrolę nad Ziemią i jej mieszkańcami . A więc do dzieła .

Uruchomienie programu i gra .

Po załadowaniu się programu komputer sprawdzi, czy jesteś fachowcem, czy jeszcze człowiekiem , którego należy podszkolić . Zada Ci pytanie - LOGON PLEASE ? Uprowadź teraz kod uwodawczy. Dla ułatwienia podajemy , że jest nim nazwa jednego z kontynentów . Gdy będziesz znał to hasło , ominie cię test robota i przejdziesz od razu do drugiego etapu . Załóżmy jednak, że nie znasz hasła , a więc na wymienione wyżej pytanie napisz **H** i **RETURN** . Komputer powiadamia was wtedy o zmianie hasła i zadaje jeszcze raz pytanie jak wyżej . Należy nacisnąć wtedy tylko **RETURN** . Gdy to wykonasz , pojawi się rysunek robota . Twoim zadaniem będzie odgadnięcie tych w robocie miejsc, których nazwy będą pojawiały się na dole ekranu . Aby przejść do następnego etapu, należy bezbłędnie odnaleźć wszystkie części. Kursor sterowany jest przy pomocy manipulatora . Po ustaleniu miejsca występowania części należy nacisnąć przycisk (FIRE) . Jeżeli odpowiedź będzie właściwa, komputer po pytaniu napisze Ci OK. i zada następne . Jeżeli natomiast pomylisz się, pytanie będzie powtarzane dotąd , aż odpowiesz prawidłowo . A ponadto test zostanie jeszcze raz powtórzony . Dla ułatwienia, pokazany został poniżej robot z umiejscowieniem poszczególnych części .

TEST ROBOTA SRU.

1. INFRARED VIDEO IMAGE SENSOR - ośrodek wizyjny
2. ASYNCHRONOUS DATA COMPACTOR - manipulator
3. HYDRAULIC MOTIVATOR - napęd
4. PHLAMSON JOINT - mechanizm kierunku
5. THELMAN PORT - gniazdo połączeń zewnętrznych



Drugi etap rozpoczyna się od pytania o twe imię.Gdy je wprowadzisz, zostaniesz poproszony o ustawienie strefy czasowej na Ziemi. Poruszając manipulatorem w lewo lub prawo ustawiasz tak porę dnia, by ci odpowiadała . Dla utrwalenia sytuacji naciskasz przycisk . Komputer informuje wtedy , że nastąpiła weryfikacja czasu . Robot SRU znajduje się w tej chwili na południowym Atlantyku . Naciśnij **RETURN** . Gdy wykonasz to polecenie , rozpoczniesz właściwą grę . Zobaczysz tablicę wskaźników (rysunek poniżej) , na której będziesz otrzymywał dane .

1. Jaki jest aktualnie wykonywany rozkaz;
2. Na monitorze obraz, jaki widzi

przed sobą robot;

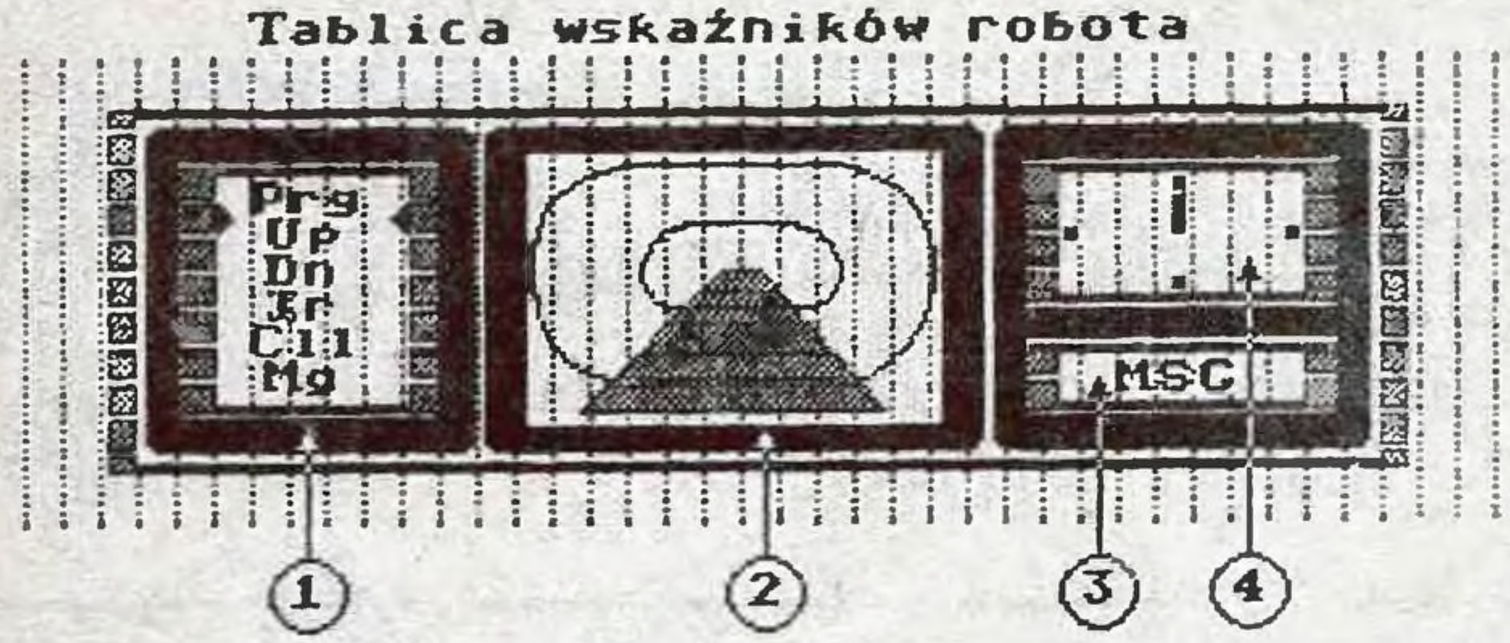
3. Sygnalizacje zgłaszania się kierownictwa akcji (migający skrót MSG);

4. Na wskaźniku określenie kierunku w jakim zwrócony jest robot.

START - powoduje powrót na początek gry

Kierowanie robotem w tunelu odbywa się za pomocą manipulatora . Po skierowaniu drążka w kierunku , w którym chcemy się poruszać , robot

Tablica wskaźników robota



Po rozpoczęciu gry z pewnością zgłosi się kierownictwo akcji wciśnij wtedy klawisz M. Otrzymasz szereg instrukcji. Spróbuj je przetłumaczyć .Może w ten sposób wzbogacisz swój słownik o nowe określenia . Jeżeli jednak nie poradzisz sobie, nie przejmuj się tym. W czasie gry będziesz miał do czynienia z dwoma grupami rozkazów . Pierwsze to te , które będziesz wydawał pod Ziemią, a mianowicie klawisz

U -oznacza wyprowadzenie robota na powierzchnię , gdy znajduje się on pod Ziemią ;

I -włącza , noktowizor gdy okazuje się, że na powierzchni jest noc ;

C - przywołuje się szpiega, by dokonać transakcji;

D -powrót robota z powrotem do tunelu ;

M - nawiązanie (na polecenie MSG) kontaktu z kierownictwem w celu otrzymania instrukcji .

Druga grupa jest związana tylko z operacjami wykonywanymi na powierzchni Ziemi (po przywołaniu szpiega) i tak naciśnięcie klawisza :

D - jak poprzednio oznacza powrót robota do tunelu ;

I - zmiana proponowanego szpiegowi przedmiotu na inny , gdy ten nie jest nim zainteresowany ,

Y - wyrażenie zgody na kupienie od szpiega przedmiotu ;

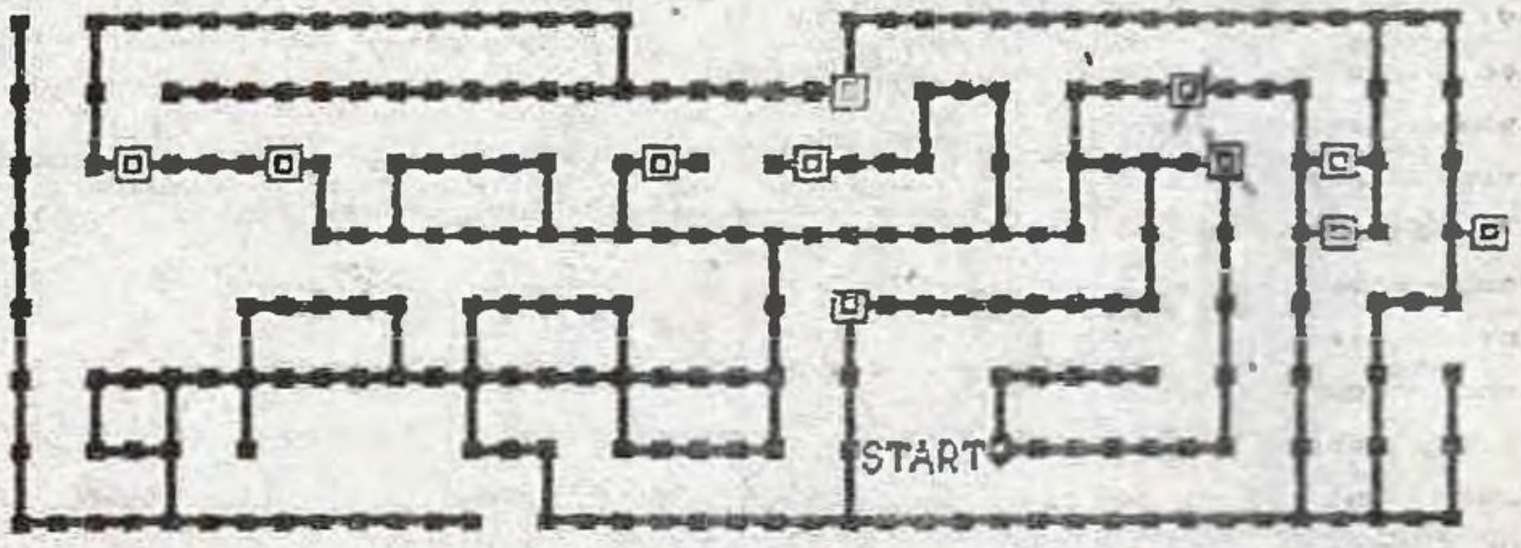
N - odmowa kupna ;

E - daje możliwość obejrzenia dotychczas wykupionych części planu;

U - jak w pierwszym rodzaju rozkazów ;

obraca się i jeżeli dalej przytrzymany , tym samym powodujemy zaprogramowanie fragmentu drogi. Gdy naciśniemy przycisk , robot przebędzie ten fragment tunelu . W zależności od tego , w którą stronę został zwrócony robot , będziesz widział dalszy ciąg tunelu lub ścianę . Schemat tunelu przedstawiono na rysunku poniżej .

Plan podziemnych korytarzy.



Załóżmy teraz , że znajdujesz się pod miastem. W takim przypadku należy nacisnąć klawisz U. Robot zostanie wyniesiony na powierzchnię . W zależności od tego , w jakiej strefie czasowej znajdujesz się , może być na powierzchni dzień (nic nie naciskasz) lub noc (wtedy musisz włączyć noktowizor klawiszem I).

Manipulatorem możesz uzyskać widok tego, co znajduje się w zasięgu widzialności robota . Po naciśnięciu klawisza E zostaje przywołany do robota szpieg. Po przyjściu proponuje ci dokonanie wymiany. Nie zawsze pytania szpiega będą zrozumiałe , gdyż z reguły będą posługiwać się swoimi " narodowymi " językami . Klawiszem I będziesz mógł pokazywać swoje oferty . Gdy szpieg zaakceptuje którąś z nich , automatycznie da ci za nią część planu (zobaczysz ją na ekranie), a gdy naciśniesz RETURN zaczniesz przedstawiać swoje propozycje do sprzedania . Przedmioty te często będą związane z krajem, w którym będziesz aktualnie się znajdował . Dla przykładu w Egipcie , szpieg zaproponuje ci szmaragdowego skarabeusza i złotą statuetkę TUT , w Grecji imitację starożytnego przedmiotu i urnę grecką , itd . Możesz je kupować , wtedy naciskasz klawisz Y lub nie (klawisz N) . Następnie przyciskając D, ponownie opuszczasz robota do tunelu i kierujesz go do następnego miasta w celu zdobycia kolejnych części planu . Przy odwiedzaniu miast powinna być zachowana pewna kolejność , gdyż szpiedzy najczęściej za plan będą żądali przedmiotów , które kupiłeś wcześniej . Odkrywając rabla tajemnicy, można powiedzieć, że np. szpieg w Anglii będzie chciał za plan chronograf, w Grecji szmaragdowego skarabeusza czy w USA (San Francisko) album z autografami Beatlesów , itd . Ze względu na ograniczony czas gry musisz sprawnie pokonywać poszczególne etapy . Jeżeli zbyt wolno będziesz zdobywać poszczególne części planu, to przeciwnik może cię uprzedzić i zakończysz swą wędrówkę. Z tego samego powodu przeciwnik może wysłać satelity szpiegowskie do unieszkodliwienia robota . Gdy zostaje on wykryty, otrzymujesz pytanie o kod rozpoznawczy . Jeśli odpowiesz źle, robot zostanie zlikwidowany . Znając ten kod, zaglisz satelitę podając robota za ich sprzymierzeńca . A oto kody w kolejności , w jakiej będą żądane przez satelity :

1. MAGMA, LTD.

2. AX-0310479

3. HYDRAULIC
4. AUSTRALIA

Krążąc po tunelu, uważaj na rejon zastrzeżony. Znajduje się on w rejonie Australii. Gdybyś przypadkiem wkroczył w jego obszar, gra skończy się ze względu na zniszczenie robota. Jak widać, zabawa nie jest taka prosta, ale czyż nie warto dowiedzieć się co kryje plan MAGMA ?

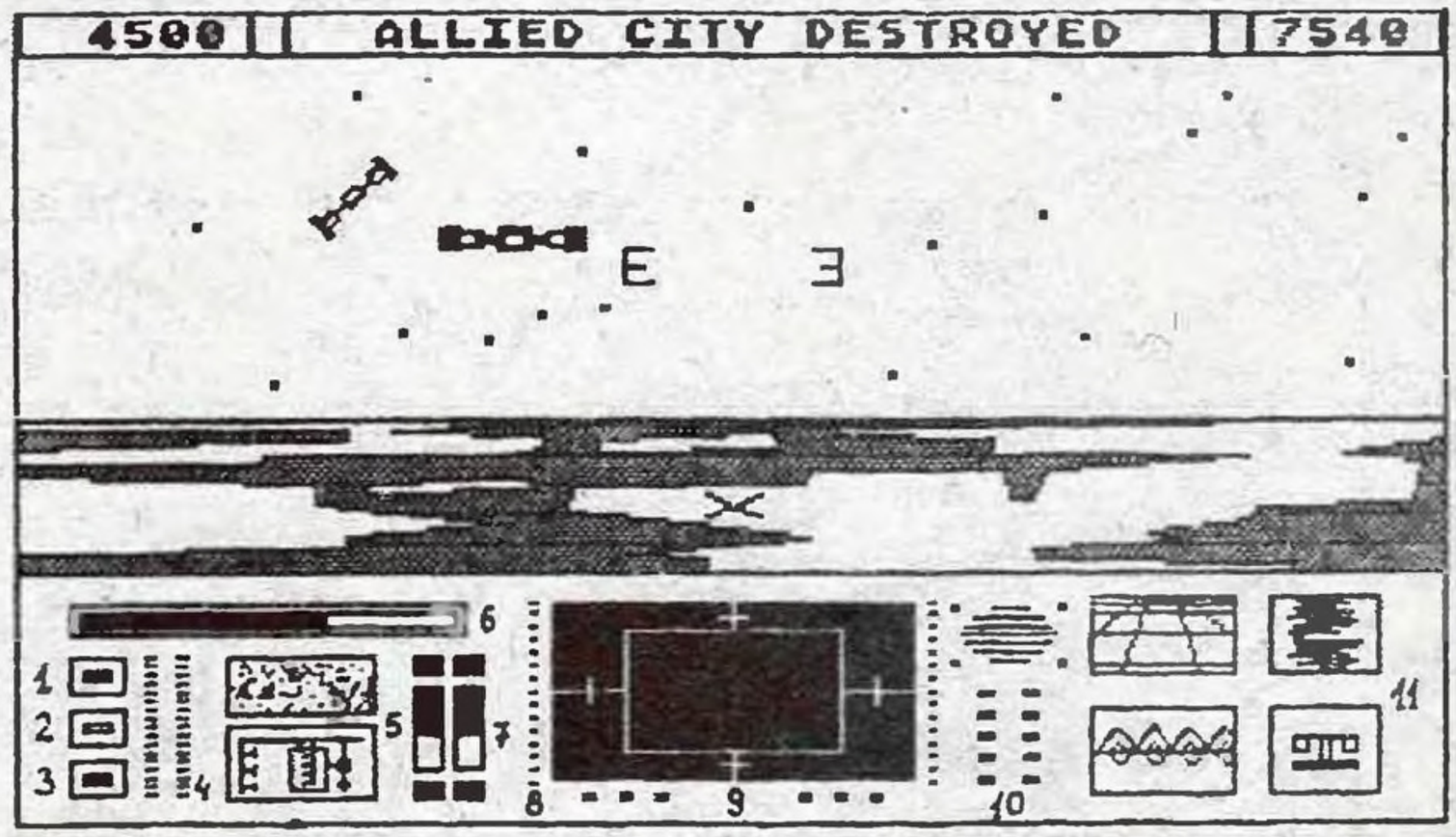
W historii istot myślących wielkie-my obrzędy walk między rodzajem ludzkim, następnie między ludźmi a mieszkańcami innych układów słonecznych, aby w czwartym tysiącleciu naszej ery doprowadzić do zawarcia pokoju w całym dotychczas zbadanym obszarze galaktyki. Doprowadziło to do eksplozji dobrobytu. Ludzie z poprzednich epok nie byliby w stanie zrozumieć tego pojęcia ani tego, że przez blisko tysiąc lat nie było konfliktów, których rozstrzygnięcie odbywałoby się przez walkę zbrojną.
Oczywiście taka sielanka nie mogła trwać wiecznie. Znalazła się w końcu grupa ludzi, która wydostała się spod kontroli unii. Utworzyli oni w układzie słonecznym XUR niezależną bazę. W tym momencie stało się jasne, że kończy się okres spokoju. Mieszkańcy układu słonecznego PROCYON, którego Ty jesteś przedstawicielem, widząc zagrożenie swojego bytu, przystąpili do budowy super statku mogącego stawić czoło ewentualnym najeźdźcom. To właśnie tobie przypadł zaszczyt go pilotować i walczyć. Po załadowaniu programu wybierz klawiszem **SELECT** stopień trudności (DIFFICULTY) od 1 do 3. Natomiast ucisnięcie **START** lub przycisku manipulatora uruchabia silniki statku wyrzucające go w przestrzeń międzyplanetarną. Od tego momentu rozpoczyna się Twoja przygoda. Jak zapewne już zauważyłeś, siedzisz przed pulpitem sterowniczym. Na dużym ekranie obserwujesz sytuację wokół najbliższego otoczenia statku. Powyżej ekranu po lewej stronie masz aktualny wynik zdobytych punktów, po prawej wartość rekordową. Środek to różnego rodzaju komentarze, z których najczęściej występujące to:

- ENEMY IN RANGE (wróg w zasięgu);
- SYSTEM MALFUNCTION (wadliwie działa system);
- ALLIED CITY DESTROYED (zniszczone zostało bronione miasto);
- BARRIER UNDER ATTACK (zaatakowana bariera układu);
- HULL TEMP CRITICAL (krytyczna temperatura kadłuba statku);
- WARP ENGINES ENGAGED (niebezpieczeństwo uszkodzenia silników);
- TRAJECTORY PLOTTED (plan układu słonecznego);
- ENERGY LEVEL CRITICAL (krytyczny poziom energii).

Poniżej ekranu znajduje się cały szereg wskaźników, a mianowicie:

1. Sygnalizator pracy działek laserowych;
2. Sygnalizator pracy działek termicznych;
3. Sygnalizator pracy działek neutronowych;
4. Wskaźnik ilości amunicji termicznej;
5. Wskaźniki ostrzegawcze;
6. Wskaźnik poziomu energii statku;
7. Wskaźnik temperatury działek laserowych;
8. Urządzenie czasowe.
9. Ekran radaru.
10. Połączenie foniczne z centrum unii.
11. Wskaźniki informujące o aktualnej sytuacji na zewnątrz statku.

Najprawdopodobniej za chwilę nadlecą pojazdy ze szwadronu KODAN-a. Poruszają się po trzy atakując trzema falami. Ich pociski nie są groźne, lecz potrafią być dokuczliwe. Twój jeden celny strzał likwiduje pojazd przeciwnika, dając 100 punktów premii. Jeżeli uda Ci się zlikwidować wszystkich atakujących, czeka Cię przeprawa z następną partią znacznie trudniejszych do zniszczenia trzech statków. Każdy musi być trafiony trzykrotnie pociskiem termicznym. Za zniszczenie statku otrzymuje się 500 punktów. Zmianę rodzaju pocisków wybiera się automatycznie. Po zniszczeniu armady należy odnaleźć następną i zniszczyć, itd. W celu przeszukania układu słonecznego wciśnij **SPACE**. Otrzymasz na ekranie obraz tego układu z zaznaczonymi aktualnymi pozycjami przeciwnika. Chcąc przenieść się w inne miejsce układu, należy manipulatorem przeciągnąć linię do punktu, w którym chcemy się znaleźć, a następnie nacisnąć przycisk. Prawidłowe ustawienie punktu docelowego zostanie potwierdzone przez powstanie otoczki wokół tego miej-

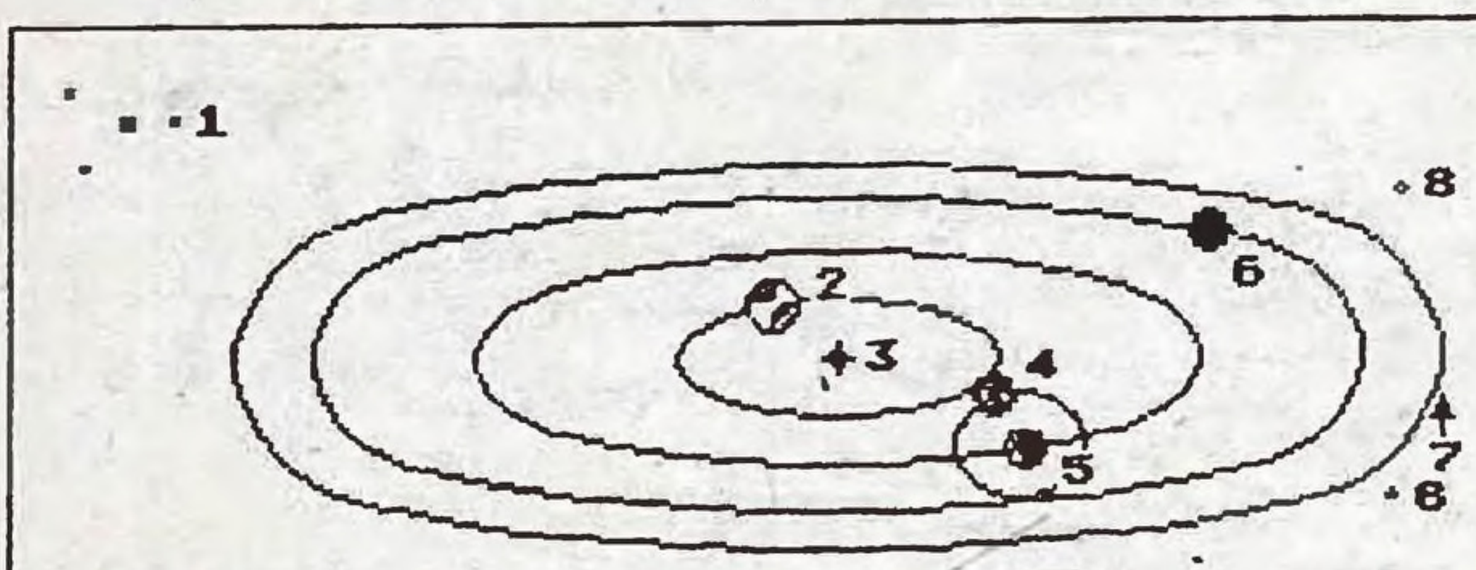


sca . Najszybciej armadę można zniszczyć , gdy przekracza ona barierę ochronną układu . Wystarczy wtedy przenieść się w to miejsce i oddać jeden strzał , a armada przestanie istnieć , za co otrzymasz 500 punktów . Na mapie układu słonecznego statki KO-DAN-a zaznaczone są w postaci trzech punktów. Gdy dojdziesz do wniosku , że pozbyłeś się wojsk przeciwnika w swoim układzie słonecznym , powinieneś zaatakować układ słoneczny XUR . Zasady walki w układzie przeciwnika są prawie identyczne jak wtedy , gdy broniłeś swojego terenu . Roznica polega jedynie na tym , że atakować Cię będą znacznie liczniejsze grupy statków. Ponadto będziesz musiał niszczyć bazy przeciwnika rozmieszczone na planetach układu . Za zniszczenie bazy otrzymujesz 1000 punktów. Bazę niszczy się pociskami neutronowymi , które można wystrzelić , gdy przełączysz rodzaj prowadzonego ognia , np. klawiszem **OPTION** .

W fervorze walki nie zapomnij o wyczerpującej się Ci energii . Gdy stwierdzisz, że posiadasz jej zbyt mało do prowadzenia dalszej walki, przenieś statek w pobliże gwiazdy układu , w którym się znajdujesz . Nie możesz jednak zbyt długo tam przebywać . Po usłyszeniu sygnału ostrzegawczego lub wcześniej postaraj się , jak najszybciej przenieść w inne miejsce układu . Gdy tego nie uczynisz , Twój statek spali się i zabawę będziesz musiał rozpocząć od nowa poprzez jednoczesne przyciśnięcie klawiszy **START** i **SELECT** . W czasie przenoszenia się otrzymujesz szereg mniej lub bardziej potrzebnych informacji o miejscu , w które chcesz się udać . Jeżeli jest to planeta , uzyskujesz jej :

- nazwę , np. KO-DAN
- kod identyfikacyjny , np. ID 805.4515.4951
- masę planety , np. M=5.3E27 g
- promień planety, np. R=5.6E08 cm
- otoczenie (ENVIRONMENT) , np. umiarkowane (TEMPERATE)
- ilość miast (CITIES) lub baz (BASES)
- liczbę statków przeciwnika (DESTROYERS) nad planetą
- liczbę pojazdów bojowych (FIGHTERS) przeciwnika nad planetą .

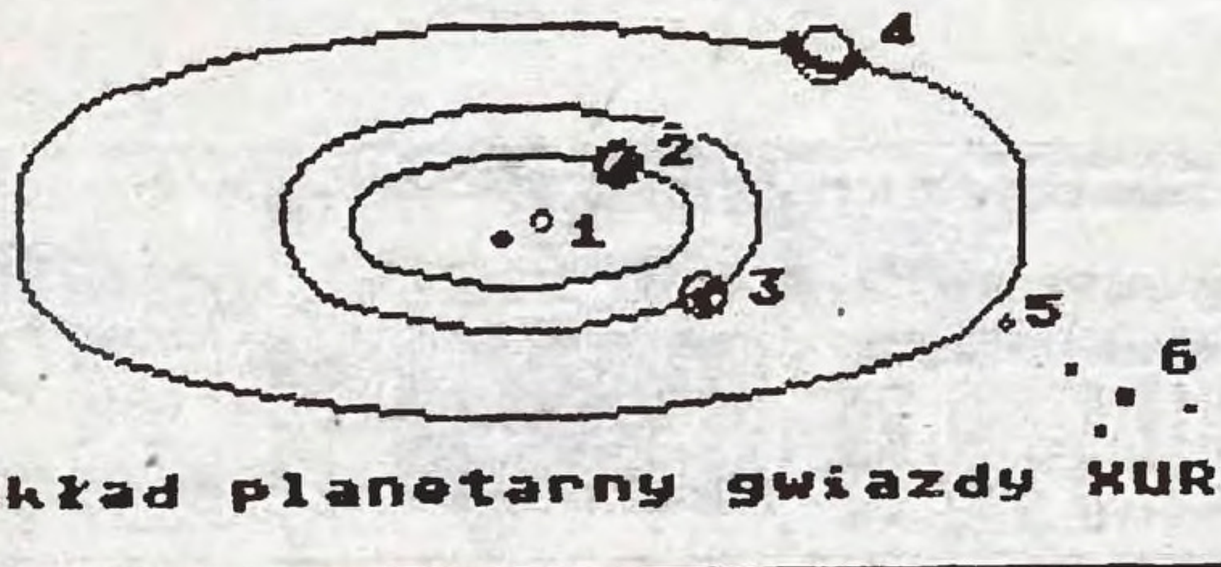
Aby zwyciężyć przeciwników z KO-DAN , powinieneś nie tylko celnie strzelać , ale wykazać się przebiegłością i pewną dozą strategicznego myślenia . Nie zawsze przewidłowe posunięcie musi w sposób zadowalający wpływać na następne decyzje . A więc walczcie i zwyciężajcie .



Układ planetarny gwiazdy Celos IV

Skład układu planetarnego Celos IV .

1. Układ planetarny XUR
2. SERIOUS
3. Słońce CELOS IV
4. GALEN
5. RYLOS
6. ARCANUS
7. Bariera ochronna układu
8. Eskadry statków z KO-DAN



Układ planetarny gwiazdy XUR

Skład układu planetarnego XUR .

1. Słońce XUR
2. GRON
3. KO-DAN
4. MORKOTH
5. Eskadry z KO-DAN
6. Układ planetarny Celos IV

- IKS - - dodatek - Żołnierza Wolności ". Redaguje Wiesław Cetera (kierownik zespołu). Rada programowa: Krzysztof Chwarra, Romuald Głąb, Włodzimierz Gogołek, Janusz Janiec, Henryk Krasuski , Ireneusz Miernik , Ludwik Piela , Jacek Szaniawski . Adres redakcji : 00-950 Warszawa ul . Grzybowska 77 , telefon centrali 20-12-61 w. 485. Telex 313680 .

Nakładów nie zamówionych redakcja nie zwraca i zastrzega sobie prawo do skrótów . Nakładem Wydawnictwa " Czasopisma Wojskowe ", Warszawa ul. Grzybowska 77. Druk offsetowy - Wojskowe Zakłady Graficzne im. gen. dyw. A. Zawadzkiego.

Nr zam 1507 Nr ind 396281 U-33