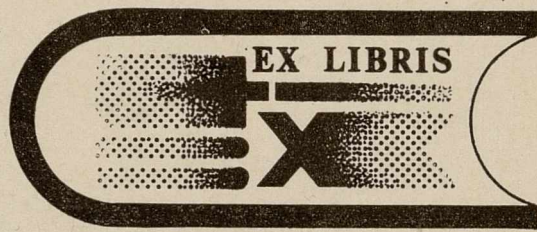


I+Z

Dariusz Król
Andrzej Siemiński

Zaawansowane metody oprogramowywania systemów informacyjnych w języku C





**BIBLIOTEKA GŁÓWNA
POLITECHNIKI WROCŁAWSKIEJ**

Politechnika Wrocławska



Dariusz Król
Andrzej Siemiński

Zaawansowane metody
oprogramowywania systemów
informacyjnych w języku C

Wrocław 1992

Wydział Informatyki i Zarządzania, Zakład Systemów Informacyjnych
Wydawnictwo dydaktyczne do przedmiotu *Programowanie systemów
informacyjnych na specjalizacji SINT, Inżynieria oprogramowania*,
kierunek Informatyka

Opiniodawcy
Zbigniew KIERZKOWSKI
Zygmunt MAZUR

Opracowanie redakcyjne
Maria KOPEĆ

Korekta
Barbara WACHOWSKA



Biblioteka Główna i OIMT
Politechniki Wrocławskiej



001052262

225332/1

© Copyright by Wydawnictwo Politechniki Wrocławskiej, Wrocław 1992

WYDAWNICTWO POLITECHNIKI WROCŁAWSKIEJ
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

ISBN 83-7085-010-3

Nakład 500+55 egz. Ark. wyd. 12. Ark. druk. 13¹/₂. Papier offset. kl. III, 70 g, B1.
Oddano do druku w lipcu 1992 r. Druk ukończono w sierpniu 1992 r.
Zakład Graficzny Politechniki Wrocławskiej. Zam. nr 483/92.

Rozdział 1

WSTEP

Celem publikacji jest dostarczenie Czytelnikowi wskazówek metodologicznych, dotyczących sposobu pisania i uruchamiania programów, przedstawienie funkcji realizujących najczęściej wykonywane zadania oraz omówienie oprogramowania narzędziowego stosowanego w oprogramowywaniu tekstowych systemów informacyjnych. Przez pojęcie tekstowe systemy informacyjne rozumiemy tutaj takie systemy, w których duża część danych jest zapisana w postaci tekstu w języku naturalnym. Systemy takie cechują się specyficznymi własnościami, z których najważniejsze to: bardzo duże rozmiary danych, ich słaba strukturalizacja oraz zmienna długość rekordów; same systemy zaś są zazwyczaj bardzo złożone. Przetwarzanie danych tekstowych jest omówione w rozdz. 6, pozostałe rozdziały zawierają informacje przydatne w tworzeniu systemów informacyjnych dowolnego typu.

Praca jest przeznaczona dla osób zajmujących się oprogramowywaniem dużych systemów informacyjnych w języku C. Podczas jej pisania zakładaliśmy, że Czytelnik zna dobrze co najmniej jeden algorytmiczny język programowania, przy czym jeżeli językiem tym nie jest język C, to do zrozumienia treści niniejszej publikacji konieczne jest wcześniejsze zapoznanie się z podstawami tego języka. Zalecamy w takim wypadku dokładne przestudiowanie podręcznika [KR].

Obecnie dostępne są opracowania na temat metodologii projektowania systemów, np. [Niedz], jeszcze więcej jest prac przedstawiających podstawy programowania, najczęściej na przykładzie konkretnego języka, w tym także na przykładzie języka C [Biel, KR]. Praca ma wypełnić lukę między tymi dwoma rodzajami opracowań. Zawiera ona omówienie zasad tworzenia oprogramowania dużych systemów, podaje przykładowe realizacje ich podstawowych bloków funkcjonalnych i omawia oprogramowanie narzędziowe, stosowane do ich tworzenia. Integralną częścią opracowania jest biblioteka zawierająca wszystkie omówione i wzmiankowane w jego treści funkcje.

Dodatkowego wyjaśnienia wymaga użyte na wstępie pojęcie dużego systemu informacyjnego. System będziemy określać jako duży, jeżeli kod źródłowy całości jego oprogramowania zajmuje co najmniej 100 KB. Oprogramowywanie takich systemów bardzo różni się od oprogramowywania pojedynczych algorytmów, podobnie jak pisanie krótkiej notatki różni się od pisania powieści. Podczas oprogramowywania algorytmu zwracamy uwagę na jego poprawność i na efektywność kodu, natomiast podczas oprogramowywania systemu nacisk jest położony na sposób integrowania jego poszczególnych części składowych. Wszelkie błędy i niekonsekwencje popełniane podczas oprogramowywania pojedynczego algorytmu można nadrobić zwiększonym nakładem pracy, poniesionym na etapie uruchamiania kodu. Rozmiar i stopień skomplikowania systemów sprawia, że podobne działania są w ich przypadku znacznie trudniejsze, a często są one wręcz niemożliwe do wykonania. Kolejna różnica polega na tym, że algorytmy są z reguły oprogramowywane przez pojedyncze osoby, systemy zaś to domena pracy zespołowej. Zasadniczo różnią je także zasady optymalizacji kodu.

Większość z przedstawionych w pracy rozważań odnosi się do systemów informacyjnych dowolnego typu. Nacisk położony został na systemy, w których przetwarzane są dane tekstowe. Spowodowane to zostało kilkoma przyczynami:

- przetwarzanie danych tekstowych staje się obecnie jednym z najważniejszych obszarów zastosowań komputerów
- na potrzeby przetwarzania tekstów nie opracowano jeszcze wystarczająco elastycznych wyspecjalizowanych języków 4. generacji i w związku z tym konieczne jest stosowanie języków algorytmicznych ogólnego przeznaczenia, np. języka C
- dostępne biblioteki dla języka C zawierają tylko część oprogramowania niezbędnego do tworzenia takich systemów, np. C-tree, B-tree koncentrują się na zapewnieniu dostępu do plików bez żadnych narzędzi do przetwarzania tekstów, obsługi ekranu czy też klawiatury.

1.1. Dlaczego wybrano język C?

Jak do tej pory, mimo intensywnych prac badawczych, nie powstał standardowy język do oprogramowywania tekstowych systemów informacyjnych. W innych dziedzinach już dość dawno pojawiły się takie standardy, np. w relacyjnych bazach danych jest nim język wywodzący się z pakietu dBaseIII. Powszechnie stosowany w systemach wyszukiwania informacji pakiet CDS/ISIS ma wiele ograniczeń [ISIS]. Próby

wykorzystania w tym celu pakietów przeznaczonych do obsługi baz danych (np. Oracle czy też Informix) kończą się zwykle niepowodzeniem. Pakiety te pozwalają na łatwe wykonanie podstawowych operacji, takich jak dodanie lub zmodyfikowanie rekordów czy indeksów. Jednakże złożone przetwarzanie tekstów jest w najlepszym przypadku nieefektywne, a często wręcz niemożliwe do uzyskania.

Jednocześnie już od kilku lat obserwuje się odchodzenie od oprogramowywania dużych systemów w językach najniższego poziomu, tj. assemblerach. Spowodowane jest to tym, że podobną sprawność kodu wynikowego oferują obecnie języki algorytmiczne, wymagające znacznie mniejszego nakładu pracy podczas tworzenia systemu. Najbardziej popularnymi językami algorytmicznymi są język Pascal i język C.

Język Pascal, opracowany w końcu lat 60. przez N. Wirtha, mimo swoich wielu zalet w porównaniu z językiem C, jest znacznie mniej odpowiedni do oprogramowywania dużych systemów. Język Pascal został opracowany jako narzędzie wspomagające nauczanie podstaw informatyki i w związku z tym jest on znakomitym narzędziem do opisu pojedynczych algorytmów. Brak jest w nim mechanizmów ułatwiających łączenie wielu takich algorytmów w system. Próby przekształcenia go w narzędzie służące do takich zastosowań doprowadzają do powstania języka mało podobnego do jego pierwowzoru. Rezultatem tego są trudności z przenoszeniem oprogramowania, a ponadto zatracą on swe podstawowe zalety: prostotę i przejrzystość. Widać to dokładnie na przykładzie kolejnych wersji implementacji języka Pascal, opracowanych przez firmę Borland.

W przeciwieństwie do języka Pascal, język C został opracowany w celu oprogramowania systemu UNIX - jednego z najbardziej złożonych systemów operacyjnych. Zalety języka C są najlepiej widoczne podczas oprogramowywania dużych systemów. Język ten ma bowiem mechanizmy ułatwiające łączenie w jedną całość modułów tworzących system. Charakteryzuje się on ponadto: nowoczesnymi strukturami danych, instrukcjami strukturalnymi, swobodą w operowaniu typami danych, różnorodnością operacji na wskaźnikach oraz bogatą i w znacznej części przenośną biblioteką funkcji (ANSI C). Jedną z najważniejszych jego zalet jest efektywność i relatywnie mały rozmiar kodu generowanego przez kompilator. Zalety tego języka spowodowały, że stosowany jest on coraz częściej do oprogramowywania pakietów (np. Clipper, Framework), wypierając stosowane poprzednio do tego celu języki assemblerowe. Język C w pełni zasługuje na miano nowoczesnego języka algorytmicznego i wydaje się, że pozostanie on w użyciu jeszcze wiele lat.

Autorzy nie przewidują wykorzystania niniejszego wydawnictwa do nauczania początkowego programowania w języku C. Zamieszczony w rozdz. 2 opis języka dotyczy tylko tych jego fragmentów, ze zrozumieniem których – jak wykazała wieloletnia praktyka – studenci mają najwięcej kłopotów. Praca jest kierowana przede wszystkim do studentów kierunków informatycznych i pochodnych, którzy opanowali już sztukę programowania w dowolnym języku strukturalnym i zetknęli się z językiem C.

1.2. Materiały źródłowe

W pracy wykorzystano doświadczenia zebrane podczas wieloletniego prowadzenia wykładu "Programowanie w języku C" oraz wykładu "Oprogramowanie systemów informacyjnych" na Wydziale Informatyki i Zarządzania Politechniki Wrocławskiej, a także doświadczenia uzyskane przez autorów podczas projektowania i oprogramowywania w języku C różnorodnych systemów informacyjnych.

Prezentowane w tej publikacji oprogramowanie jest częściowo naszego autorstwa, częściowo zaś zaczerpnęliśmy je z innych publikacji. Najwięcej takich funkcji pochodzi z opracowania *Advanced C Programming for Displays* [Roch], tj. funkcje realizujące trzy pierwsze warstwy obsługi ekranu i dwie warstwy obsługi klawiatury.

Do innych prac szczególnie pomocnych w trakcie pisania tego opracowania i zawierających rozszerzenia materiału w nim zamieszczonego należy znakomita publikacja *Algorithms in C* [ALGinC] oraz dokumentacje kompilatorów języka C, opracowanych przez firmy Borland [BC1-5] i Microsoft [MC].

Autorzy pragną złożyć serdeczne podziękowania recenzentom za cenne uwagi, które pozwoliły na usunięcie wielu pomyłek i błędów. Podziękowania składamy również studentom Wydziału Informatyki i Zarządzania specjalności SINT, a zwłaszcza Krzysztofowi Janiszewskiemu, za pomoc w testowaniu oprogramowania i uwagi dotyczące treści pracy.

1.3. Przegląd treści

Praca składa się z 9 rozdziałów.

Rozdział 1 zawiera wstęp.

Rozdział 2 poświęcono prezentacji wybranych elementów języka C. Nacisk położono na te z nich, które są ważne podczas pisania dużych systemów. W nielicznych wypadkach opisano skrótowo zagadnienia dość proste, takie jak np. rodzaje zmiennych i funkcji, operacje na wskaźnikach, czy też zarządzanie pamięcią. Doświadczenia zebrane podczas prowadzenia zajęć dydaktycznych wskazują jednak, że pojęcia te nie zawsze są należycie rozumiane przez studentów. Oprócz tego bardzo szczegółowo omówiono preprocesor i sposoby jego wykorzystania, a także przedstawiono zestaw makr do obsługi plików.

W rozdziale 3 omówiono zasady pisania wielowarstwowych, wielomodułowych systemów. W proponowanym sposobie pisania oprogramowania system pod względem logicznym jest podzielony na warstwy. Pod względem fizycznym oprogramowanie jest natomiast podzielone na moduły. Opisywane w tym rozdziale zagadnienia obejmują zasady tworzenia oprogramowania wielowarstwowego, zalecane sposoby kodowania funkcji, propozycje struktury plików z programami źródłowymi oraz sposoby zapewniania przenośności i optymalizacji kodu.

Kolejne dwa rozdziały omawiają oprogramowanie ekranu i klawiatury. Użytkownik komunikuje się za ich pośrednictwem z systemem i dlatego właściwy sposób obsługi klawiatury i ekranu ma bardzo duże znaczenie w ocenianiu jakości pracy systemu. Prezentowane oprogramowanie ekranu można traktować jako wzorcowy przykład tworzenia oprogramowania wielowarstwowego. Składa się ono z 4 warstw: ekranu fizycznego, okien, ekranów wirtualnych i funkcji użytkowych. Analogicznie oprogramowanie klawiatury składa się z 3 warstw, są nimi: klawiatura fizyczna, klawiatura wirtualna i funkcje użytkowe klawiatury. Pozwalają one na bardzo elastyczną obsługę klawiatury, np. na definiowanie własnych makr czy też zmianę interpretacji klawiszy.

Rozdział 6 poświęcono przetwarzaniu tekstów w języku naturalnym. Przedstawiono w nim sposoby rozwiązywania podstawowych problemów z tej dziedziny. Należą do nich: przechowywanie, opisywanie i wyszukiwanie tekstów o zadanych własnościach. Omówiono także sposoby przyspieszania przeszukiwania tekstu przez stosowanie szybkich algorytmów szukania wzorca, a także stosowanie pliku inwersyjnego, jak i badanie tekstowego podobieństwa słów, co pozwala z kolei na zwiększenie kompletności wyszukiwania.

Rozdział 7 przeznaczono na prezentację oprogramowania wspomagającego. Obejmuje on omówienie wyspecjalizowanych edytorów przeznaczonych dla programistów, programów nadzorujących tworzenie kodu wynikowego, obsługujących biblioteki i wyszukujących w plikach wystąpień zadanego ciągu znaków. Nacisk położono na zasady

wykorzystania profilerów do optymalizacji kodu. Na zakończenie rozdziału omówiono typowe błędy popełniane podczas pisania programów w języku C.

Załączniki umieszczone na końcu pracy zawierają skrótowy opis wszystkich funkcji zamieszczonych w bibliotece CTLS i liczne tabele z informacjami często wykorzystywanymi przez programistę, takimi jak np. wykaz najczęściej wykorzystywanych funkcji edytorów programowych.

1.4. Sposób prezentowania oprogramowania

Źródłowa postać omawianego w pracy oprogramowania zajmuje ponad 130 KB. Szczegółowe omówienie całości oprogramowania w skrypcie nie jest zatem możliwe ani nawet konieczne - Czytelnicy mogą bezpłatnie otrzymać bibliotekę z wszystkimi omawianymi funkcjami. W opracowaniu tym dokładnie przedstawiono jedynie sposób wywołania i działania wszystkich funkcji zewnętrznych. Dodatkowo przedstawiono kod źródłowy kilkunastu funkcji wraz z jego analizą. Omawiane w ten sposób są przede wszystkim funkcje wyróżniające się zastosowaniem trudniejszych, rzadziej stosowanych konstrukcji języka lub też ilustrujący standardowy w danej sytuacji sposób postępowania. Funkcje te są wykorzystane w 3 programach demonstracyjnych.

Bibliotekę zawierającą wszystkie omawiane funkcje można skopiować w sekretariacie Zakładu Systemów Informacyjnych, pok. 203e, budynek A1 Politechniki Wrocławskiej, tel. 20-32-58. Oprócz tego można tam też skopiować kod źródłowy wszystkich tych funkcji, które są dokładnie omówione w treści pracy, a także pliki nagłówkowe i programy demonstracyjne. Biblioteka jest utworzona z funkcji skompilowanych w modelu Small za pomocą licencjonowanej wersji kompilatora Borland C++ v2.0.

Rozdział 2

WYBRANE ELEMENTY JEZYKA C

Publikacja ta nie jest podręcznikiem programowania w języku C. Przedmiotem niniejszego rozdziału nie jest ani opis tego języka, ani też omówienie standardowych funkcji bibliotecznych. Przytoczone tutaj definicje i przykłady zostały wybrane z dwu powodów. Pierwszy z nich to chęć podkreślenia cech wyróżniających język C spośród innych języków programowania, a zwłaszcza odróżniających go od powszechnie znanego języka Pascal. Drugi wynika bezpośrednio z charakteru publikacji. Szczegółowo omówiono te elementy języka C, na które zwracać trzeba uwagę podczas oprogramowywania dużych systemów.

2.1. Własności zmiennych i funkcji

Język C został zaprojektowany do oprogramowywania dużych systemów. Takie przeznaczenie języka uwidacznia się na przykład w rodzajach dostępnych w nim zmiennych i funkcji. Zostały one zaprojektowane w taki sposób, by ułatwić konstruowanie systemu z opracowywanych niezależnie fragmentów.

Kompilator po napotkaniu w wyrażeniu zmiennej lub funkcji musi mieć o nich pewne informacje po to, aby wygenerować kod realizujący to wyrażenie. Informacje te może on uzyskać z ich definicji lub deklaracji. Definiowanie zmiennej obejmuje określenie jej nazwy, typu, zakresu obowiązywania oraz sposobu przydzielenia jej pamięci. Podczas jej deklarowania wystarczy podać tylko trzy pierwsze elementy. Definiowanie służy kompilatorowi do utworzenia zmiennej (przydzielenia jej pamięci), deklarowanie zaś do sprawdzenia poprawności jej użycia. W przypadku funkcji definicja obejmuje podanie jej nazwy, typu, zakresu obowiązywania, typów parametrów i treści. W przypadku deklaracji wymagane jest podanie jedynie nazwy i typu. Zakres obowiązywania i typy parametrów, choć nieobowiązkowe, powinny jednak być zaznaczone - pozwala to bowiem kompilatorowi na dokładniejsze sprawdzenie poprawności wyrażenia. Definicja zmiennej lub funkcji jest obowiązkowa. Występuje ona tylko raz w całym programie. Deklaracji

może być dowolnie wiele. Język C zezwala nawet na ich pominięcie. W pewnych przypadkach nie ma to żadnego wpływu na przebieg kompilacji programu, lecz przyjęte w rozdz. 3 zasady pisanía programów wymagają, aby każdorazowe użycie zmiennej lub funkcji było poprzedzone jej zdefiniowaniem lub deklaracją. Pozwala to na uniknięcie wielu trudnych do wykrycia błędów wykonania.

2.1.1. Zmienne

Zmienne różnią się typem, zakresem ich obowiązywania i sposobem przydzielania im pamięci.

Język C dysponuje wszystkimi podstawowymi typami zmiennych występującymi w językach algorytmicznych, a także mechanizmem tworzenia przez użytkownika na ich podstawie typów pochodnych.

Pod względem zakresu obowiązywania rozróżniamy zmienne wewnętrzne i zewnętrzne. Zmienne wewnętrzne są definiowane lub deklarowane na początku dowolnego bloku (blokiem nazywamy fragment kodu źródłowego ograniczony znakami '{' i '}'). Zmienne zewnętrzne są definiowane lub deklarowane poza jakimkolwiek blokiem. Zakresem obowiązywania zmiennych wewnętrznych jest zawsze blok, w jakim występują. Zmienna wewnętrzna x przesłania wszystkie zmienne zewnętrzne x oraz wszystkie zmienne wewnętrzne z bloków zawierających dany blok, mające taką samą jak ona nazwę. Zakresem obowiązywania zmiennych zewnętrznych jest cały plik, począwszy od miejsca ich zdefiniowania lub zadeklarowania.

Domyślnie wszystkie zmienne wewnętrzne mają pamięć przydzielaną dynamicznie, tzn. ze stosu. Możliwe jest jednak przydzielenie takiej zmiennej pamięci statycznie (w segmencie danych, tak by jej położenie nie zmieniał się w trakcie wykonywania programu), lub też przechowywanie jej w rejestrze. W pierwszym przypadku należy jej typ poprzedzić słowem kluczowym *static*, a w drugim słowem *register*. Zastosowanie słowa kluczowego *register* odniesie pożądany skutek tylko wtedy, gdy będzie dostępny wolny rejestr na zapamiętanie takiej zmiennej. Oczywiście można w ten sposób przydzielić pamięć jedynie zmiennym, które można zapamiętać w jednym rejestrze, nie jest możliwe więc zapamiętanie w ten sposób tablicy. Natomiast w celu przyspieszenia szybkości wykonywania programu można zapamiętać w ten sposób wskaźnik na tablicę. Zmienne zewnętrzne otrzymują pamięć zawsze statycznie.

Początkujący programiści wykazują często znaczną niefrasobliwość w określaniu rodzaju używanych zmiennych i nie dostosowują rodzaj zmiennych do zadań przez nie spełnianych. Częstym źródłem nieporozumień jest używanie słowa kluczowego *static* zarówno do określenia sposobu przydzielania pamięci, jak i zakresu obowiązywania zmiennej. Umieszczenie go przed definicją zmiennej wewnętrznej sprawia, że (zgodnie z intuicją) pamięć dla niej zostanie przydzielona statycznie. Zapisanie go przed definicją zewnętrzną oznacza dodatkowo, że zakresem obowiązywania tej zmiennej jest jedynie plik, w jakim ją zdefiniowano. Jeżeli chcemy w innych plikach skorzystać ze zmiennej, to jej definicja nie może zawierać słowa *static*, a jej użycie musi być poprzedzone jej zadeklarowaniem. Deklaracja zmiennej ma postać definicji poprzedzonej słowem kluczowym *extern*.

Różnice w użyciu zmiennych lokalnych, którym przydziela się pamięć statycznie i dynamicznie, występują głównie w funkcjach wywoływanych rekurencyjnie i podczas ich inicjowania. Zmienne rejestrowe opisano dokładniej w p. 2.3.2.

Poniższy przykład ilustruje sposoby definiowania różnych rodzajów zmiennych (litera S oznacza pamięć przydzielaną statycznie D zaś - dynamicznie).

Plik a.c:

```
int dla_programu;          /* S, zakresem obowiązywania jest
                             cały plik, a potencjalnie program */
static int dla_pliku;      /* S, zakres obowiązywania
                             ograniczony do pliku a.c */

int f()
{
    register int do_rejestru; /* D, zmienna przeznaczona do
                                umieszczenia w rejestrze */
    int lokalna;             /* D, lokalna dla funkcji f */
    static int statyczna;    /* S, lokalna dla funkcji f */

    while (1)
    {
        int lokalna;        /* D, lokalna dla instrukcji while,
                                przesłania zmienną lokalną
                                funkcji f */

        int i;              /* D, lokalna dla instrukcji while */
        ...
    }
}
```


Plik b.c:

```
extern int dla_pliku;      /* deklaracja zmiennej, która musi
                           być zdefiniowana w innym pliku */

main()
{
  register int r;          /* D, zmienna przeznaczona do
                           umieszczenia w rejestrze */
  r=dla_pliku;
}
```

2.1.2. Funkcje

W języku C istnieje znacznie mniej rodzajów funkcji niż zmiennych. Wynika to z tego, że wszystkie funkcje otrzymują pamięć statycznie, a ponadto niedozwolone jest definiowanie funkcji wewnątrz innych funkcji. Domyślnie każda funkcja jest dostępna w całym programie. Jeśli stosowanie jej chcemy ograniczyć do jednego pliku, to jej definicję należy poprzedzić słowem kluczowym *static*. Kompilator zezwala nawet na korzystanie z niezadeklarowanych funkcji. Nie należy jednak korzystać z tego udogodnienia, ponieważ nie jest wówczas sprawdzana poprawność jej stosowania (poprawność użytego typu funkcji i typu jej parametrów), a ponadto kompilator zakłada, że typem funkcji jest *int*, co może prowadzić do błędów wykonania, np. w funkcjach zwracających wskaźniki, patrz p. 7.6.3.

Deklarację funkcji tworzy jej nagłówek, po którym następuje znak *;*. Definicja funkcji utworzona jest z nagłówka, po którym następuje treść funkcji.

Poniższy przykład ilustruje sposoby definiowania różnych rodzajów funkcji.

plik c.c:

```
static char *robocza(char *s); /* deklaracja funkcji lokalnej dla
                               pliku c.c */

int badaj(char *a, char *b)    /* początek definicji funkcji badaj */
{
  char *w=robocza(a);          /* takie inicjowanie zmiennej w jest
                               możliwe dzięki deklaracji funkcji
                               robocza */
  ...
}                               /* koniec definicji funkcji badaj */

static char *robocza(char *s) /* początek definicji funkcji robocza*/
```

```

{
...
}                                /* koniec definicji funkcji robocza */

plik d.c:
int badaj (char *a, char *b); /* deklaracja funkcji badaj */
main()
{
    int wynik;
    wynik=badaj("asd","cdf");
    ...
}

```

2.2. Preprocesor

Preprocesor przetwarza kod źródłowy programu przed rozpoczęciem pracy kompilatora. Preprocesor działa zatem niezależnie od kompilatora. Można go wykorzystać jako prosty program do przekształcania tekstów - kompilatory pozwalają na zatrzymanie przetwarzania pliku przed rozpoczęciem jego właściwej kompilacji [BC1]. Dyrektywy preprocesora, choć nie należą bezpośrednio do standardu języka, znacznie rozszerzają środowisko programowe, a umiejętne ich stosowanie znacznie ułatwia pracę programisty. Dyrektywy preprocesora obowiązują od miejsca ich wystąpienia w pliku do jego końca, jeżeli inna dyrektywa nie anuluje ich działania.

Preprocesor C, pracujący zgodnie z normą ANSI C, zawiera następujące dyrektywy: **#define**, **#error**, **#include**, **#if**, **#else**, **#elif**, **#endif**, **#ifdef**, **#ifndef**, **#undef**, **#line**, **#pragma**.

Wszystkie dyrektywy rozpoczynają się znakiem hash '#'. Jeden wiersz programu może zawierać jedną dyrektywę. Aby zwiększyć czytelność programu, zaleca się zapisywanie jej na początku wiersza.

2.2.1. #define

Dyrektywa **#define** jest używana do definiowania identyfikatora i ciągu znaków, który jest podstawiany w miejsce występowania identyfikatora w kodzie źródłowym. Identyfikator często jest nazywany makrem. Ogólnie definicja ta wygląda następująco:

```
#define nazwa_makra ciąg_znaków
```

Nie zaleca się kończenia makr znakiem ';', może to bowiem prowadzić do powstania trudnych do usunięcia błędów kompilacji. Jeśli tylko ciąg znaków jest za długi, by zmieścić się w wierszu, należy przed znakiem nowej linii umieścić znak backslash '\', oznaczający kontynuację

wiersza. Raz zdefiniowane makro może być użyte w innym makrze. Kompilator Borland niestety nie spełnia wszystkich wymogów normy ANSI C i nie zastępuje makra, jeśli znajdzie się ono w ciągu, np.

```
#define TEKST to jest ciąg
printf("TEKST");
```

nie zostanie wypisany ciąg "to jest ciąg", lecz napis "TEKST".

Dyrektwy **#define** powinno używać się z reguły dla niewielkich bloków instrukcji, wskazane jest też używanie ich w pętlach. Popularne stało się wśród programistów stosowanie dużych liter jako identyfikatorów i umieszczanie wszystkich definicji w jednym miejscu, najlepiej w osobnym pliku nagłówkowym. Jednym z najczęstszych zastosowań jest definiowanie za pomocą makr wszelkich stałych. Zamiast pisania liczb określających liczbę iteracji, czy wielkość tablicy, lepiej jest zdefiniować makro. W ten sposób późniejsza zmiana wielkości wymaga korekty programu tylko w jednym miejscu zamiast w każdym miejscu jego użycia.

Ciąg występujący w tej dyrektywie może zawierać blok instrukcji wykonujący działanie pewnej funkcji. Zastąpienie wywołań funkcji przez tak zdefiniowane makro może znacznie zwiększyć szybkość działania programu. Zysk polega na uniknięciu odkładania na stosie adresu powrotu i ewentualnych argumentów wywołania, co osiągnąć jest oczywiście kosztem rozmiaru kodu (duplikacja instrukcji).

Bardziej rozbudowane postacie dyrektywy **#define** mają (podobnie jak funkcje) parametry, na przykład po zdefiniowaniu:

```
#define MIN(x,y) (x<y) ? x : y

wywołanie

printf ("minimum to %d", MIN(a,b));
```

spowoduje zamianę powyższego wiersza przez preprocesor na:

```
printf("minimum to %d",(a<b) ? a : b);
```

Należy jednak zwrócić szczególną uwagę na kolejność wykonywania operacji podczas definiowania makr, np. w wypadku stosowania pozornie poprawnego makra **PARZYSTE**:

```
#define PARZYSTE(x) x%2==0 ? 1 : 0
```

Wywołania typu **PARZYSTE(9+1)** da nam wynik błędny. Preprocesor zamieni bowiem to makro na ciąg **9+1%2==0 ? 1 : 0**. Wyrażenie to ma wartość 0, bowiem operator **%** ma wyższy priorytet niż operator dodawania **+**. Poprawna wersja makra **PARZYSTE** ma postać:

```
#define PARZYSTE(x) (x)%2==0 ? 1 : 0
```

Niebezpieczeństwo stosowania operatorów inkrementacji i dekrementacji w makrach opisano w p. 3.3.1.

Na koniec przedstawiamy zestaw makr zapewniających uniwersalną, lecz jednocześnie prostą implementację obsługi tablic o wielkości ustalonej w trakcie wykonywania programu. Makra te pozwalają na tworzenie i kasowanie takich tablic, a także na zapisywanie i pobieranie poszczególnych elementów z tych tablic. W przykładzie elementami tymi są jednowymiarowe tablice znaków i liczby całkowite. Nic jednak nie stoi na przeszkodzie, by makra te zastosować np. do struktur czy unii. Inny bardziej skomplikowany przykład stosowania makr znajduje się w p. 2.4.

```
#include <alloc.h>
#include <mem.h>

#define INI(w,typ,il) w=calloc(sizeof(typ),il)
#define END(w) if (w!=NULL) free(w);
#define PUT(w,el,ktory) if(w!=NULL) \
    memmove(w+ktory,&el,sizeof(el))
#define GET(w,ktory,gdzie) if (w!=NULL) \
    memmove(&gdzie,w+ktory,sizeof(gdzie))
#define ZERUJ(x) memset(&x,0,sizeof(x))
#define PODSTAW(wsk,wartosc) if (wsk!=NULL) *wsk=wartosc

typedef char tab_zn[6];

main()
{
    tab_zn t1="34567", t2="abc", *wt;
    int i1=111, i2=222, *wi;
    int x=5, y=7;

    wi=NULL;
    PODSTAW(wi,7);      /* podstawienie nie wykona się */
    wi=&i2;
    PODSTAW(wi,7);      /* podstawienie wykona się */
    ZERUJ(i1);          /* równoważne i1=0 */
    ZERUJ(t1);          /* równoważne t1="" */

    INI(wt,tab_zn,10); /* alokacja 10 tablic 6-elementowych */
    PUT(wt,t2,0);      /* skopiowanie t2 do pierwszej tablicy */
    PUT(wt,t1,2);      /* skopiowanie t1 do trzeciej tablicy */
    GET(wt,0,t1);      /* skopiowanie pierwszej tablicy do t1 */
    END(wt);           /* zwolnienie 10 tablic 6-elementowych */
}
```

```
INI(wi,int,5);      /* tak jak wyżej, tyle że operujemy na liczbach */  
PUT(wi,i1,0);  
PUT(wi,i2,1);  
GET(wi,0,i2);  
END(wi);  
}
```

Zainteresowani Czytelnicy już we własnym zakresie mogą napisać makra do obsługi wielowymiarowych tablic. Przydatne do tego celu mogą być uwagi zamieszczone w [Schil].

2.2.2. #error

Dyrektywa **#error** powoduje zatrzymanie kompilacji tuż po jej napotkaniu. Ogólna definicja jest następująca:

#error tekst_informacyjny

Tekst_informacyjny nie jest umieszczany w cudzysłowie. Po napotkaniu tak zdefiniowanej dyrektywy proces kompilacji jest przerywany i wyświetlany jest komunikat:

Fatal: nazwa_pliku numer_wiersza

Error directive: tekst_informacyjny.

2.2.3. #include

Dyrektywa **#include** dołącza inny plik do pliku, z którego została ona wywołana. Nazwa pliku włączanego musi zostać ujęta w nawiasy <> lub w cudzysłów "". Różnica polega na tym, że w pierwszym przypadku włączany plik jest poszukiwany w kartotekach wyspecyfikowanych podczas konfiguracji środowiska, natomiast w drugim przypadku najpierw w bieżącej kartotece, następnie w kartotekach podanych w linii zlecenia i dopiero na końcu tak, jak w określonych przez środowisko [BC1]. Jeśli jawnie wyspecyfikowano ścieżkę dostępu do pliku, to tylko ta kartoteka jest przeszukiwana. Dozwolone jest stosowanie zagnieżdżonych dyrektyw. Niedozwolona jest jednak rekurencja, tzn. sytuacja, w której np. z pliku A wczytujemy plik B, a z B plik A.

2.2.4. #if, #else, #elif, i #endif

Wymienione dyrektywy są ze sobą ściśle powiązane. Służą one do kompilacji warunkowej. Jeśli wyrażenie występujące po **#if** jest spełnione, to kod pomiędzy tą dyrektywą i **#endif** jest kompilowany. Jeśli nie, to kod ten jest pomijany. Dyrektywa **#endif** zaznacza koniec bloku rozpoczętego przez **#if**. Ogólna postać jest następująca:

```
#if wyrażenie
    lista_rozkazów
#endif
```

Ponieważ wyrażenie jest obliczane podczas kompilacji, więc może zawierać tylko wyrażenia poprzednio zdefiniowane lub stałe. Nie można używać zmiennych.

Dyrektywa `#else` działa podobnie jak `else` w instrukcji `if`. Jest alternatywa, gdy `#if` nie jest spełnione. Ponadto `#else` spełnia jeszcze jedną rolę. Zaznacza zarówno koniec bloku `#if`, jak i początek bloku `#else`. Jest to konieczne, gdyż tylko jeden `#endif` może być powiązany z każdym `#if`.

Dyrektywa `#elif` ma znaczenie `else if` i używana jest podobnie jak dyrektywa `#if`. Uogólniony zapis jest następujący:

```
#if wyrażenie
    lista_rozkazów
#elif wyrażenia 1
    lista_rozkazów
#elif wyrażenia 2
    lista_rozkazów
...
#elif wyrażenia N
    lista_rozkazów
#endif
```

W Borland C++ można używać operatora `sizeof` po dyrektywie `#if`. Poniższy przykład pokazuje jak w niekonwencjonalny sposób można określić model pamięci, w którym kompilowany jest program.

```
#if (sizeof(char *) == 2)
    printf("Model Small, Tiny lub Medium");
#else
    printf("Model Compact, Large lub Huge");
#endif
```

Takie użycie tej dyrektywy jest jednak niezgodne z normą ANSI C.

2.2.5. `#ifdef`, `#ifndef` i `#undef`

Innym sposobem uzyskania warunkowej kompilacji jest użycie `#ifdef` (jeśli zdefiniowano) lub `#ifndef` (jeśli nie zdefiniowano). Jeżeli makro zostało zdefiniowane za pomocą `#define` lub wymienione w linii zlecenia kompilatora (`-Dxxx` dla Borlanda lub `/D xxx` dla Microsofta), to lista rozkazów pomiędzy `#ifdef` i `#endif` jest uwzględniona w

kompilacji. Dyrektywa `#ifndef` ma znaczenie przeciwne. Obie te dyrektywy mogą używać dyrektywy `#else`, ale nie mogą używać dyrektywy `#elif`. Podobnie jak w przypadku `#if` zagnieżdżanie jest dopuszczalne. Ostatnia z wymienionych, dyrektywa `#undef`, usuwa poprzednio zdefiniowane makro. Stosowanie jej pozwala na posiadanie lokalnych definicji.

Znajomość preprocesora ułatwia programowanie. W dostarczanych wraz z kompilatorem standardowych zbiorach nagłówkowych nagminnie stosuje się dyrektywy preprocesora, dzięki czemu zachowana jest przejrzystość definicji, prototypów funkcji, typów, struktur i unii oraz uniwersalność ich stosowania. Jako przykład niech posłuży fragment zbioru nagłówkowego `stdio.h`.

```
#ifndef __STDIO_H          /* definiowanie __STDIO_H */
#define __STDIO_H

#if __STDC__              /* definiowanie _Cdecl */
#define _Cdecl
#else
#define _Cdecl cdecl
#endif

#ifndef __PAS__           /* definiowanie _CType */
#define _CType _Cdecl    /* dla niepascalowego typu */
#else
#define _CType pascal    /* dla pascalowego typu */
#endif

#ifndef NULL              /* definiowanie NULL */
#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
#define NULL 0            /* dla Tiny, Small i Medium */
#else
#define NULL OL          /* dla Compact, Large i Huge */
#endif
#endif

/* definicje makr: getchar i putchar */
#define getchar() getc(stdin)
#define putchar(c) putc((c), stdout)
#endif
```

Dzięki kompilacji warunkowej można uzyskać z jednego pliku źródłowego wiele plików wynikowych. Zmiana parametrów kompilacji pozwala na wygenerowanie np. wersji użytkowej, demonstracyjnej lub rezydentnej tego samego programu. Wykorzystanie kompilacji warunkowej podczas testowania programu omówiono w p. 2.2.2.3.

Stosowanie kompilacji warunkowej do pomijania fragmentów kodu podczas kompilacji jest znacznie lepsze niż tworzenie z tego fragmentu kodu komentarza. Kompilacja warunkowa nie wymaga bowiem ingerencji w kod źródłowy, a ponadto unika się kłopotów z zagnieżdżonymi komentarzami.

2.2.6. #line

Dyrektywa **#line** zmienia zawartość predefiniowanych makr **__LINE__** i **__FILE__**. Makro **__LINE__** zawiera numer wiersza kompilowanego, natomiast **__FILE__** nazwę kompilowanego pliku. Ogólna postać **#line** to:

```
#line liczba ["nazwa_pliku"],
```

gdzie liczba jest dowolną liczbą naturalną, a nazwa_pliku ciągiem znaków dla pliku. Dyrektywa ta może być stosowana podczas debuggowania do śledzenia działania programu. Oczywiście metoda ta nie zastąpi pomocy oferowanej przez profesjonalny debugger, patrz p. 7.6.6. Oto prosty przykład:

```
#line 100 "przykład.c"           /* zamiana zawartości */
main()                          /* wiersz 100 */
{                                /* wiersz 101 */
printf("%d%s", __LINE__, __FILE__); /* wiersz 102 plik przykład.c */
}
```

2.2.7. #pragma

Dyrektywa **#pragma** znacznie rozszerza możliwości języka C. Ogólna postać jest następująca:

```
#pragma dyrektywa_implementacyjna.
```

Dyrektywa implementacyjna może wystąpić z ewentualnymi parametrami. Kompilator Borland C++ 2.0 dysponuje licznymi dyrektywami implementacyjnymi. Poniżej znajduje się krótki opis dla każdej z nich:

argsused

Jest dozwolona tylko między definicjami funkcji i oddziałuje tylko na następną funkcję. Swoim działaniem dezaktywuje komunikat:

"Parameter <nazwa> is never used in function <nazwa_funkcji>".

```
exit <nazwa_funkcji> [priorytet]
```

```
startup <nazwa_funkcji> [priorytet]
```

Pozwalają one specyfikować funkcje, które powinny być uruchomione przed wywołaniem funkcji **main** lub przed zakończeniem programu, czyli bezpośrednio przed wywołaniem funkcji **_exit**. Nazwa_funkcji musi być

poprzednio zadeklarowaną, bezargumentową funkcją, zwracającą void, np. void funkcja(void); musi wystąpić przed wierszem z **#pragma**. Priorytet jest opcjonalną liczbą z zakresu <0,255>, przy czym 0 oznacza najwyższy priorytet. Wartość 100 jest domyślna. Nie powinno się używać wartości z przedziału <0,63>, ponieważ są one zastrzeżone dla standardowych funkcji bibliotecznych. Funkcje z wyższym priorytetem są wywoływane jako pierwsze przy wejściu do programu (**#pragma startup**) lub jako ostatnie na zakończenie działania (**#pragma exit**).

hdrfile

Przedefiniowuje nazwę pliku, w którym pamiętane są prekompilowane zbiory nagłówkowe. Domyślną nazwą jest TCDEF.SYM. Przykładowe zastosowanie: **#pragma hdrfile "nazwa_pliku.SYM"**.

hdrstop

Zaznacza koniec dla listy zbiorów nagłówkowych w procesie prekompilacji. W ten sposób ulega zredukowaniu zajmowane miejsce na dysku przez prekompilowane zbiory nagłówkowe. Dokładniejszy opis procesu prekompilacji znajduje się w p. 3.3.5 i 3.3.6.

saveregs

Gwarantuje, że w wyniku wywołania funkcji nie zmieni się zawartość żadnego z rejestrów procesora. Dyrektywę tę należy umieścić bezpośrednio przed definicją takiej funkcji.

option <opcje...>

Daje możliwość bezpośredniego włączenia opcji z linii zlecenia do programu źródłowego. Można stosować dowolne opcje z wyjątkiem:

-B	-c	-Dxxx=ccc
-dxxx	-E	-efff
-Fx	-H	-lfff
-Lfff	-lxset	-M
-o	-P	-Q
-S	-T	-Uxxx
-V	-X	-Y

Wymienione poniżej opcje mogą pojawić się tylko przed **#if**, **#ifdef**, **#ifndef**, **#elif** i przed pierwszą deklaracją:

-Efff	-f*	-i#
-hddd	-m*	-offf
-u	-W	-z*

dowolna opcja zmiennoprzecinkowa z wyjątkiem **-ff**
dowolna opcja modelu pamięci

Następujące opcje oddziałują na kompilator, jeżeli zostały użyte między funkcjami: -1 -2 -a -ff -G -h -k -N -O -p -r -rd -v -y -z. Natomiast opcje: -A -b -C -d -gn -jn -K -wxxx -zE -zF -zH mogą być stosowane dowolnie i efekt ich działania jest bezpośredni.

inline

Informuje kompilator o występowaniu w kodzie źródłowym instrukcji asm.

warn +xxx, -xxx, .xxx

Powoduje odpowiednio (+) włączenie, (-) wyłączenie lub (.) przełączenie ostrzeżenia xxx w procesie wyświetlania wyłapanych przez kompilator ostrzeżeń. Dozwolonymi wartościami ostrzeżeń xxx są:

Ostrzeżenia standardu ANSI

bei Inicjacja z niepoprawnym typem
 bfs Pole bitowe rzutowane na liczbę całkowitą ze znakiem
 big Stała szesnastkowa lub ósemkowa za duża
 dup Redefinicja nieidentyczna
 eas Przypisanie <typ int> do <typ enum>
 ext <Identyfikator> zadeklarowany jako zewnętrzny i statyczny
 ill Zle zdefiniowana pragma
 pin Inicjacja częściowo wzięta w nawiasy
 ret Używany return i return wartość
 str Poza struktura
 stu Niezdefiniowana struktura
 sus Podejrzana konwersja wskaźnika
 voi Funkcja typu void nie może zwracać wartości
 zdi Dzielenie przez zero
 zst Struktura o długości zerowej

Częste błędy

aus Przypisana wartość nigdy nie używana
 def Możliwe użycie przed definicją
 eff Instrukcje bez wpływu na program
 par Nie używany parametr
 pia Możliwość niepoprawnego przypisania
 rch Nieosiągalny kod
 rvl Funkcja powinna zwracać wartość

Najczęstsze błędy

amb Dwuznaczne operatory wymagają nawiasów
 amp Niepotrzebny & dla funkcji lub tablicy
 nod Brak deklaracji dla funkcji
 pro Wywołanie funkcji bez prototypu
 stv Struktura przekazywana przez wartość
 use Zadeklarowany, lecz nigdy nie użyty

Przenośność

apt Nieprzenośne przypisanie wskaźnika
 cln Stała typu long
 cpt Nieprzenośne porównanie wskaźnika
 rng Stała spoza zakresu przy porównaniu
 rpt Nieprzenośna konwersja wskaźnika
 sig Konwersja "gubi" bity znaczące
 ucp Pomieszczenie wskaźników na char "ze znakiem" i "bez znaku"

2.2.8. Predefiniowane makra kompilatora

Standard ANSI C specyfikuje pięć wbudowanych makr (patrz p. 2.2.1):

Makro	Typ	Znaczenie
<code>__LINE__</code>	int	numer bieżącego wiersza
<code>__FILE__</code>	char *	nazwa pliku
<code>__DATE__</code>	char *	bieżący miesiąc/dzień/rok
<code>__TIME__</code>	char *	czas od rozpoczęcia kompilacji (h:min:s)
<code>__STDC__</code>	int	1, jeśli jest to standardowa implementacja ANSI C

Natomiast Borland C++ definiuje dodatkowe makra:

<code>__BCPLUSPLUS__</code>	stała równa 0x0200, jeśli tylko wybrano kompilację C++
<code>__BORLANDC__</code>	numer wersji: 0x200 dla BORLAND C++ 2.0
<code>__CDECL__</code>	=1, jeśli konwencja wywoływania jest ustawiona na język C
<code>__COMPACT__</code>	zdefiniowane po wybraniu modelu pamięci Compact
<code>__cplusplus</code>	=1, jeśli w trybie C++
<code>__DLL__</code>	=1, jeśli moduł jest kompilowany z opcją -WD
<code>__HUGE__</code>	zdefiniowane po wybraniu modelu pamięci Huge
<code>__LARGE__</code>	zdefiniowane po wybraniu modelu pamięci Large
<code>__MEDIUM__</code>	zdefiniowane po wybraniu modelu pamięci Medium
<code>__MSDOS__</code>	=1
<code>__OVERLAY__</code>	=1, jeśli moduł kompilowany z opcją -Y
<code>__PASCAL__</code>	=1, jeśli konwencja wywoływania jest ustawiona na Pascal
<code>__SMALL__</code>	zdefiniowane po wybraniu modelu pamięci Small
<code>__TCPLUSPLUS__</code>	numer wersji: 0x200 dla BORLAND C++ 2.0

<u>TINY</u>	zdefiniowane po wybraniu modelu pamięci Tiny
<u>TURBOC</u>	zdefiniowane jako 0x0297
<u>WINDOWS</u>	zdefiniowane, jeśli moduł kompilowany jest z opcją -W

2.3. Zarządzanie pamięcią

W języku C najważniejszymi, ale i najbardziej niebezpiecznymi operacjami są działania związane z jawną alokacją pamięci. Praktyka wskazuje, że ponad 90% trudnych do wykrycia błędów jest spowodowanych niewłaściwym gospodarowaniem pamięcią. Pamięć można przydzielić na dwa sposoby: domyślnie - podczas definiowania zmiennych (patrz p. 2.1) oraz jawnie - przez użycie odpowiednich funkcji. Ponieważ pierwszy sposób jest znany także z innych języków programowania, zajmijmy się dokładniej drugim z nich.

Jawne przydzielanie pamięci jest określane często jako jej dynamiczna alokacja. Nie należy mylić jej ze zmiennymi, które dynamicznie otrzymują pamięć. Pamięć dla zmiennych jest dostarczana automatycznie przez programy i pochodzi ona ze stosu lub segmentu danych dla zmiennych statycznych, zaś dynamiczna alokacja pamięci jest wykonywana przez programistę (stąd jej nazwa - jawna) i pochodzi ona z dalekiej (far heap) lub bliskiej sterty (near heap).

Do dynamicznej alokacji służą, w zależności od używanego modelu pamięci, funkcje `malloc` (`farmalloc`) lub `calloc` (`farcalloc`). Zwalnianie danego obszaru wymaga wywołania funkcji `free` (`farfree`). W modelach Compact, Large i Huge użycie `malloc` (`calloc`, `free`) jest równoważne użyciu `farmalloc` (`farcalloc`, `farfree`). Zwolnienie obszaru powoduje dopisanie go do listy bloków z niewykorzystaną pamięcią sterty. Podczas zwalniania pamięci automatycznie łączone są przylegające bloki, zaś podczas jej alokowania optymalizowane jest miejsce, w jakim jest ona przydzielona. Jak widać, zarówno jawne przydzielenie, jak i zwolnienie pamięci jest operacją dość złożoną. Jeżeli program wykonuje wiele operacji tego typu na blokach o różnych wielkościach, to prowadzi to do znacznego spowolnienia jego pracy. Spowodowane jest to koniecznością przeglądania długiej listy wolnych bloków i jest określane jako segmentacja (fragmentacja) obszaru sterty. W takich wypadkach zalecane jest ujednolicenie wielkości bloków, patrz [BC2].

Sprawdzenie liczby bajtów między zaalokowanym najwyżej w pamięci obszarem a wierzchołkiem sterty (ang. heap), tj. końcem dostępnej pamięci, dokonuje się po wywołaniu funkcji standardowej `coreleft` (`farcoreleft`). Funkcje te nie zwracają zatem całkowitej wielkości wolnej pamięci, ale wielkość ostatniego bloku opisanej listy.

Ostatnią z funkcji służących do jawnego operowania pamięcią jest **realloc**. Powala ona zarówno na zmniejszenie, jak i zwiększenie zaalokowanego poprzednio obszaru. Przepisuje ona zawartość starego obszaru do nowego i zwalnia stary obszar. Wielkość przepisywanego obszaru nie przekracza naturalnie rozmiarów nowo otrzymanego bloku.

Każda funkcja alokacji zwraca wskaźnik na przydzielony obszar, w przypadku błędu jest to **NULL**. Następstwa niesprawdzenia takiej funkcji opisano w p. 7.6.2.

Powyżej wspomniano o modelach pamięci. Ze względu na budowę procesorów rodziny xxx86, wyróżniamy sześć różnych modeli:

Model Tiny

W modelu tym wszystkie rejestry segmentowe zawierają tę samą wartość i wszelkie operacje adresowe używają jedynie 16 bitów. To oznacza, że kod, dane i stos muszą pomieścić się w jednym segmencie (64 KB). Oczywiście wytworzony kod jest najmniejszy i najszybszy. Może być typu .EXE lub .COM.

Model Small

Jest to model domyślny, wystarczający w większości zastosowań. Adresowanie jest nadal 16-bitowe, tak jak w Tiny. Kod odseparowano od danych i stosu, stąd dopuszczalna wielkość to 128 KB.

Model Medium

Jest przeznaczony dla dużych programów, w których kod przekracza 64 KB. Jednakże stos i dane nie mogą przekroczyć 64 KB. To rozwiązanie jest szczególnie przydatne w dużych programach, korzystających z danych o stosunkowo niewielkich rozmiarach.

Model Compact

Model ten jest komplementarny do modelu Medium. W tym przypadku kod jest ograniczony do 64 KB, natomiast dane mogą zajmować kilka segmentów.

Model Large

W tym modelu kod i dane są ograniczane w zasadzie tylko przez dostępną wolną pamięć. Całkowita wielkość danych statycznych nie może jednak przekroczyć 64 KB. Domyślnie wszystkie wskaźniki i funkcje są typu far.

Model Huge

Model Huge przejmuję wszystkie własności modelu far z jednym wyjątkiem – wskaźniki są typu huge. Pozwala to na wykorzystanie bez ograniczeń całej dostępnej pamięci. Wadą tego modelu jest fakt, że wygenerowany kod jest najdłuższy i wykonuje się najdłużej spośród wszystkich modeli.

Kompilator Microsoft C pozwala ponadto programiście na definiowanie własnych modeli pamięci [MC].

Tak jak wspomniano, najczęściej używa się modelu Small. Czasami jednak potrzebujemy zamiast 16-bitowego adresu, pełny adres 20-bitowy, np. by uzyskać bezpośredni dostęp do ekranu. Rozwiązaniem jest stosowanie modyfikatorów pamięci: *far*, *near* i *huge*. Można je stosować także do funkcji. I tak:

far - umożliwia dostęp poza bieżący segment danych, jednakże podczas operacji porównania czy inkrementacji (dekrementacji) tylko mniej znaczący bajt (offset) jest porównywany czy aktualizowany.

near - narzuca 16-bitowe traktowanie wskaźnika, jest to offset w DS. Stosowany jest on w funkcjach rekurencyjnych.

huge - zachowuje się jak modyfikator *far* z tym, że w tym przypadku także segment (bardziej znaczący bajt) jest brany pod uwagę.

2.4. Obsługa plików

Standard ANSI C definiuje tylko jeden zbiór funkcji wejścia/wyjścia (I/O). Jednakże standard UNIX zawiera jeszcze drugi równoległy system operacji I/O. Pierwsza metoda, występująca w obu standardach, zwana jest potocznie metodą buforowaną lub wysokiego poziomu. Natomiast druga na odwrót - niebuforowaną lub niskiego poziomu.

Przed przedstawieniem przykładowego systemu I/O należy zrozumieć różnicę pomiędzy podejściem buforowanym i niebuforowanym oraz różnicę między plikiem i strumieniem.

System buforowany, w którym automatycznie utrzymywane są bufony, został zaprojektowany z myślą o bezproblemowej współpracy z różnorodnymi urządzeniami zewnętrznymi wraz z terminalami, napędami dysków itd. Chociaż każde z tych urządzeń jest różne, to zostaje ono transformowane na logiczne urządzenie zwane strumieniem. Mamy dwa rodzaje strumieni: tekstowe i binarne. Pierwszy z nich to ciąg znaków zorganizowanych w wiersze kończące się znakiem nowej linii ('\n') z możliwością translacji niektórych znaków (zależnie od implementacji). Natomiast drugi, binarny, jest ciągiem bajtów. W każdym programie automatycznie jest zdefiniowanych pięć strumieni tekstowych: *stdin*, *stdout*, *stderr*, *stdprn* i *stdaux*.

W języku C wszystkie strumienie są takie same, natomiast pliki - nie (terminal nie ma możliwości dostępu swobodnego, jaką ma plik dyskowy). Plik nieformalnie można zdefiniować jako dowolne zewnętrzne urządzenie wejścia/wyjścia. Podczas operacji otwarcia strumień jest wiązany z wyspecyfikowanym plikiem i wskaźnik pliku jest ustawiany na jego początek (wartość 0). Po przeczytaniu lub zapisaniu znaku wskaźnik ten jest inkrementowany. Operacja zamknięcia strumienia uwalnia plik i w przypadku systemu buforowanego cała zawartość bufora zostaje zapisana na dysk. Każdy strumień jest powiązany z plikiem za pomocą struktury kontrolnej typu FILE. Definicja tej struktury i prototypy funkcji operujących na strumieniach znajdują się w stdio.h. Natomiast prototypy funkcji niskiego poziomu (programista sam musi zadbać o buforowanie) znajdują się w io.h.

Rozdział ten proponuje jednolity system makr do obsługi plików z możliwością wyboru jednego standardu dostępu. System ten pozwala na utworzenie **CREATE**, otwarcie **OPEN**, otwarcie tylko do odczytu **OPENRD**, zamknięcie **CLOSE**, odczyt **READ**, zapis **WRITE**, przesunięcie głowicy **SEEK**, sprawdzenie położenia głowicy **TELL** i określenie długości pliku **FILELENGTH**. Jest on uniwersalny dla kompilatorów, ponieważ zarówno Borland C, jak i Microsoft C, używają obu systemów. Przyjęto następujące oznaczenia:

- *f* oznacza wskaźnik na FILE (ANSI C) lub liczbę typu int (UNIX)
- *var* jest wskaźnikiem na czytany/zapisywany obszar
- *ile* określa liczbę bajtów
- *pos* to liczba bajtów do przesunięcia głowicy dla **SEEK** lub wskaźnik do liczby long w przypadku **TELL** i **FILELENGTH**
- *from* oznacza pozycję startową: początkową, końcową lub bieżącą
- *txt* jest nazwą pliku.

Należy pamiętać, iż zarówno zarządzanie pamięcią, tak i zarządzanie plikami musi być kontrolowane. Kontrola ta została w porównaniu z innymi językami programowania znacznie ułatwiona, np. w języku Pascal dopiero po wywołaniu dodatkowej funkcji dowiadujemy się o błędzie wykonania funkcji typu I/O. W języku C wystarczy sprawdzić wartość zwracaną z funkcji. W poniższym przykładzie w przypadku wystąpienia błędu zostanie wykonana instrukcja return i zostanie zwrócona informacja o błędzie. Wobec tego każda funkcja używająca jednego z tych makr musi to uwzględniać. Dokładny opis używanych funkcji znajduje się w pracy [KR].

Standard ANSI C - buforowany

```

#define READ(f,var,ile) \
    if (fread((char *)var, ile, 1, f) != 1) return(ERR)
#define WRITE(f,var,ile) \
    if (fwrite((char *)var, ile, 1, f) != 1) return(ERR)
#define SEEK(f,pos,from) \
    if (fseek(f, (long)(pos), from) != NULL) return(ERR)
#define TELL(f,pos) \
    if ((* (long *)pos=ftell(f)) == (long)EOF) return(ERR)
#define FILELENGTH(f,pos) \
    if ((* (long *)pos=filelength(fileno(f))) == (long)EOF) return(ERR)
#define OPEN(f, txt) \
    if ((f = fopen(txt, "r+b")) == NULL) return(ERR)
#define OPENRD(f, txt) \
    if ((f = fopen(txt, "rb")) == NULL) return(ERR)
#define CREATE(f, txt) \
    if ((f = fopen(txt, "w+b")) == NULL) return(ERR)
#define CLOSE(f) if (fclose(f) == EOF) return(ERR)
#define CLOSEALL if (fcloseall() == EOF) return(ERR)

```

Standard UNIX - niebuforowany

```

#define READ(f,var,ile) \
    if (read(f, (char *)var, ile) == EOF) return(ERR)
#define WRITE(f,var,ile) \
    if (write(f, (char *)var, ile) == EOF) return(ERR)
#define SEEK(f,pos,from) \
    if (lseek(f, (long)(pos), from) == (long)EOF) return(ERR)
#define TELL(f,pos) \
    if ((* (long *)pos=tell(f)) == (long)EOF) return(ERR)
#define FILELENGTH(f,pos) \
    if ((* (long *)pos=filelength(f)) == (long)EOF) return(ERR)
#define OPEN(f, txt) \
    if ((f = open(txt,(int)(O_RDWR | O_BINARY))) == EOF) return(ERK)
#define OPENRD(f, txt) \
    if ((f = open(txt,(int)(O_RDONLY | O_BINARY))) == EOF) \
        return(ERR)
#define CREATE(f, txt) \
    if ((f = open(txt,(int)(O_CREAT|O_TRUNC|O_RDWR|O_BINARY), \
        (unsigned)(S_IRREAD|S_IWRITE))) == EOF) return(ERR)
#define CLOSE(f) if (close(f) == EOF) return(ERR)
#define CLOSEALL #error Brak takiej możliwości

```

2.5. Zmienna liczba argumentów funkcji

Język C pozwala na stosowanie funkcji o zmiennej liczbie argumentów. Najpopularniejszymi przykładami takich funkcji, ze zmienną liczbą argumentów, są `printf()` i `scanf()`.

Język C pozwala na określenie sposobu zapamiętywania argumentów funkcji na stosie. Domyślnie każda funkcja jest typu `_Cdecl`, to znaczy parametry są odkładane kolejno na stos z lewej strony na prawą. Odwrotna kolejność odkładania jest przyjęta po umieszczeniu przed nazwą funkcji słowa kluczowego `_pascal` lub `_fortran`. Teraz odbywa się to kolejno z prawej strony na lewą. Taka konwencja jest używana właśnie w tych językach programowania. Pozwala to na proste dołączenie np. procedur i funkcji napisanych w języku Pascal do programu napisanego w języku C lub na odwrot.

Znajomość tej zasady pozwala także programiście na definiowanie własnych funkcji mających zmienną liczbę parametrów. Jest kilka sposobów ich obsługi. Zaletą przedstawionego poniżej rozwiązania jest to, że zapewnia on przenośność programu na inne systemy operacyjne. Opiera się on na wykorzystaniu makr `va_start`, `va_end` i `va_arg`:

```
void va_start(va_list arg_ptr, last_parm),
void va_end(va_list arg_ptr),
type va_arg(va_list arg_ptr, type),
```

których definicje w standardzie ANSI C znajdują się w `stdarg.h`. Wersje makr kompatybilnych z UNIX'em SYSTEM V są zamieszczone w `varargs.h`.

Obie wersje makr przyjmują, że funkcja ma z góry określoną liczbę stałych argumentów, po której następuje opcjonalna, zmienna liczba argumentów. Te opcjonalne argumenty są dostępne przez makra, które znajdują wskaźnik do pierwszego parametru z listy, udostępniają kolejne argumenty i zwalniają wskaźnik po skończeniu operacji. Makro `va_start` inicjuje wskaźnik `<arg_ptr>`, `va_arg` najpierw pobiera wartość wskazanego typu spod adresu `<arg_ptr>`, a następnie inkrementuje wskaźnik `<arg_ptr>` używając `sizeof(type)`, by wskazywać na następny argument. Po uzyskaniu wszystkich argumentów makro `va_end` zwalnia `<arg_ptr>`. Sposób korzystania z tych makr ilustruje przykład:

```
/* Zdefiniuj symbol UNIX dla wersji UNIX */
#include <stdio.h>
#if !defined( UNIX )           /* wersja ANSI */
    #include <stdarg.h>
    int sum( int pierwszy, ... );
```

```

#else                                     /* wersja UNIX */
    #include <varargs.h>
    int sum( va_list );
#endif

void main()
{
    printf( "Suma wynosi: %d\n", sum( 2, 3, 4, -1 ) );
    printf( "Suma wynosi: %d\n", sum( -1 ) );
}

#if !defined( UNIX )                     /* ANSI */
int sum( int pierwszy, ... )
{
    int sum = 0, i = pierwszy;
    va_list marker;
    va_start( marker, pierwszy ); /* Inicjacja */
    while( i != -1 )
    {
        sum += i;
        i = va_arg( marker, int );
    }
    va_end( marker );                  /* Skasowanie */
    return(sum);
}
#else                                     /* UNIX */
int sum( va_alist )
va_dcl
{
    int i, sum;
    va_list marker;
    va_start( marker );                /* Inicjacja */
    for( sum = 0; (i = va_arg( marker, int )) != -1; )
        sum += i;
    va_end( marker );                  /* Skasowanie */
    return(sum);
}
#endif

```

Funkcje

```

int vfprintf(FILE *strumień, char *format, va_list argptr),
int vprintf(char *format, va_list argptr),
int vsprintf(char *buf, char *format, va_list argptr),

```

są funkcjami formatującymi dane do strumienia, standardowego wyjścia i bufora. Są one podobne do ich odpowiedników `fprintf`, `printf` i `sprintf`, ale każdy z nich akceptuje wskaźnik na listę argumentów `<argptr>` zamiast listy argumentów, które są konwertowane i wyświetlane zgodnie z wyspecyfikowanym formatem w `<format>`. Argument `<format>` ma taką samą strukturę jak w funkcji `printf`. Wynikiem funkcji `vprintf` i `vsprintf` jest liczba napisanych znaków bez kończącego zera, natomiast wynikiem funkcji `vfprintf` jest liczba zapisanych w pliku znaków. W razie wystąpienia błędu pojawia się na wyjściu wartość EOF. Poniższy program ilustruje zastosowanie funkcji `vprintf`.

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

int p_m( char *, ... );

void main()
{
    p_m( "Pierwszy parametr wynosi: %d, drugi: %d\n", 27, 13);
}

int p_m( char *format, ... )
{
    va_list marker;
    va_start( marker, format );    /* Inicjacja */
    vprintf( format, marker );
    va_end( marker );              /* Skasowanie */
}
```

Rozdział 3

ZASADY OPROGRAMOWYWANIA SYSTEMÓW INFORMACYJNYCH

Początkujący projektanci i programiści często zapominają, że autorami dużych systemów są zwykle grupy programistów, a ponadto systemy te są używane przez wiele lat, w trakcie których są wielokrotnie poprawiane i modyfikowane przez różnych ludzi. Narzuca to konieczność pisania kodu źródłowego w taki sposób, by mógł być on łatwo zrozumiany przez dowolną osobę o odpowiednim doświadczeniu. Zasadę tę opłaca się stosować nawet podczas pisania oprogramowania wyłącznie na własne potrzeby - Czytelnicy z własnego doświadczenia sami z pewnością wiedzą jak szybko umykają z pamięci szczegółowe rozwiązania przyjęte w trakcie programowania. Stosowanie jednolitych zasad nawet wtedy, gdy nie są one idealne, jest lepsze od zastępowania ich dobrą wolą pracowników.

Omawiane zasady pisania programów obejmują prezentację zalecanej struktury fizycznej i logicznej oprogramowania. W dalszej części opisano sposoby optymalizacji kodu i zapewnienia jego przenośności.

Przed przejściem do szczegółowych zaleceń, na zakończenie części wstępnej, przypomnimy kilka ogólnych zasad pisania oprogramowania.

1. Stosowanie funkcji zaczerpniętych z bibliotek radykalnie skraca czas poświęcony na pisanie oprogramowania. Przed przystąpieniem do kodowania programu należy dokładnie przeanalizować zawartość dostępnych bibliotek i wybrać z nich wszystkie przydatne funkcje.
2. Kod funkcji powinien być napisany w ten sposób, by dana funkcja realizowała nie tylko te zadania, do rozwiązania których została napisana, ale też inne im pokrewne. Pozwoli to na tworzenie bibliotek uniwersalnych funkcji.
3. Optymalizowanie kodu funkcji przed jej uruchomieniem jest stratą czasu. Wydłuża bowiem tylko czas uruchamiania, komplikuje kod źródłowy, a może nawet okazać się, że jest w ogóle zbędne - optymalizowany fragment może być wykorzystywany sporadycznie.

Pomijając przypadki trywialne, selekcji funkcji do optymalizacji można dokonać dopiero po analizie danych dostarczonych przez profiler, patrz p. 7.5.

4. Należy zachować ostrożność przy definiowaniu zmiennych statycznych w funkcjach, funkcja powinna być wykonywana poprawnie także przy jej wywołaniach rekurencyjnych.
5. Na koniec pozwalamy sobie zamieścić uwagę trywialną ale, jak to wynika z naszego doświadczenia dydaktycznego, konieczną. Nie należy przystępować do oprogramowania funkcji o ile nie znamy dokładnego sposobu jej działania. Mikrokomputery są czasami obciążane winą za rozpowszechnienie się niestaranego, nieprzemyślanego sposobu programowania. Duża ich dostępność, szybkość kompilacji i satysfakcja odnoszona z faktu przystąpienia do rozwiązywania zadania tj. pisania kodu, skłania do redukcji czasu "marnotrawionego" na projektowanie. W rezultacie powstają funkcje trudne do uruchomienia i do późniejszego wykorzystania.

3.1. Struktura logiczna oprogramowania

Każdy system realizuje wiele zadań, takich jak np. obsługa urządzeń zewnętrznych, komend użytkownika itp. W zalecanym sposobie pisania kodu pod względem logicznym oprogramowanie każdego z tych zadań jest utworzone z jednej lub wielu warstw. Warstwą nazywamy tu zestaw funkcji realizujących określone zadanie na pewnym stopniu szczegółowości. Funkcje najniższej warstwy spełniają ponadto inne bardzo ważne zadanie - są one jedynymi funkcjami, jakie kontaktują się z systemem operacyjnym i wykorzystywanym sprzętem. Podczas przenoszenia kodu do nowego środowiska modyfikowana jest jedynie ta najniższa warstwa oprogramowania. Kolejne warstwy pozwalają na coraz to łatwiejsze wykonywanie najczęściej spotykanych zadań. Cena, jaką płacimy za stosowanie funkcji z wyższych warstw, jest zwiększanie się rozmiarów kodu wynikowego, wykorzystywanej pamięci operacyjnej oraz czasu wykonania programu i zmniejszenie wszechstronności stosowania tych funkcji. Uzyskane korzyści jednak je rekompensują. Są nimi: znaczne zmniejszenie czasu pisania, uruchamiania i modyfikowania systemu oraz skrócenie długości kodu źródłowego.

Warstwy tworzące oprogramowanie standardowych zadań, takich jak np. obsługa ekranu, są pisane tylko raz. Pozwala to na znaczne skrócenie czasu pisania kolejnych aplikacji. Podczas oprogramowywania konkretnego systemu wybieramy tę warstwę, która gwarantuje efektywne rozwiązanie zadania w tym systemie.

Powyższe rozważania zilustrowane zostaną przykładem oprogramowania wielowarstwowego omawianego w rozdz. 4, poświęconym obsłudze ekranu. Oprogramowanie to tworzy cztery warstwy.

Pierwsza z nich, nazwana ekranem fizycznym, pozwala jedynie na wykonanie podstawowych operacji, takich jak wyświetlanie znaków z określonymi atrybutami w zadanym miejscu ekranu. Stosowanie tych funkcji jest bardzo uciążliwe, podstawową ich zaletą jest to, że są one jedynymi funkcjami kontaktującymi się bezpośrednio z systemem operacyjnym i sprzętem. Funkcje z wszystkich pozostałych warstw stosują je do wyświetlania znaków. Przeniesienie kodu np. z IBM/AT 486, wyposażonego w kartę SVGA na terminal UNIX-a, wymaga ingerencji jedynie w kilku dość prostych funkcjach tej warstwy.

Kolejną warstwę nazywamy oknami. Pozwala ona na podzielenie ekranu fizycznego na wiele prostokątnych fragmentów i wykonywanie na nich prostych operacji, takich jak np. wpisywanie danych, ramek, zmianę rozmiarów okien itp. Wykorzystanie możliwości, jakich dostarcza ta warstwa, jest już znacznie prostsze niż dla ekranu fizycznego. Po to jednak, by w pełni z nich skorzystać, użytkownik musi dostarczyć funkcję odtwarzającą odsłaniane fragmenty okien (patrz p. 4.4). Jest to w wielu wypadkach dość uciążliwe.

Warstwa ekranów wirtualnych dostarcza standardową funkcję do odtwarzania zawartości okna. Dodatkowo warstwa ta pozwala na stosowanie wielu ekranów wirtualnych o rozmiarach potencjalnie przekraczających rozmiary ekranu fizycznego. Udogodnienia te są okupione znacznym zwiększeniem rozmiarów wykorzystywanej pamięci operacyjnej i w wielu wypadkach korzystniejsze jest stosowanie mniej uniwersalnych, ale zarazem i mniej pamięciochłonnych, funkcji.

Ostatnia z omawianych warstw (funkcji użytkowych) upraszcza wykonanie na ekranach wirtualnych najczęściej stosowanych operacji takich jak np. edycja wiersza czy wybór elementu z menu.

W celu ułatwienia zidentyfikowania warstwy, z jakiej pochodzi dana funkcja, zaleca się, aby nazwy funkcji jednej warstwy zaczynały się identycznym ciągiem znaków pisanych wielkimi literami. Dzięki temu można bez problemu określić, z jakiej warstwy pochodzą następujące funkcje wypisujące ciąg znaków: PSwrite (physical screen - ekran fizyczny), Wwrite (window - okno) czy też VSwrite (virtual screen - ekran wirtualny).

Jak widać, funkcje z wszystkich tych warstw realizują w zasadzie to samo zadanie: wyświetlanie znaków na ekranie, jednak możliwości, jakie dają one programiście, różnią się diametralnie. Wprowadzenie warstw zapewnia uporządkowanie oprogramowania i ułatwia wybór sposobu obsługi ekranu właściwego dla konkretnej aplikacji.

3.2. Struktura fizyczna systemu

Język C ma swobodną strukturę. Toleruje on na przykład umieszczanie definicji funkcji w plikach nagłówkowych, albo na odwrót - zupełne zaniechanie stosowania plików nagłówkowych. Nakłada to wielką odpowiedzialność na programistę, bowiem jego zadaniem jest wykorzystanie w sensowny sposób możliwości języka. Stosowanie omawianych poniżej zasad prowadzi zwykle do wydłużenia kodu źródłowego. Jest to jednak opłacalne w dłuższym okresie. O szybkości zakończenia prac w bardzo niskim stopniu decyduje czas pisania i wprowadzania kodu źródłowego. Przedstawione zasady nie uwolnią od popełniania błędów, pomogą jednak pisać systemy łatwe do uruchomienia i modyfikacji oraz do przenoszenia na inne komputery. Nie oznacza to też, że stosowanie ich jest obligatoryjne. Istnieje wiele przypadków, w których lepiej jest przyjąć odmienne rozwiązanie. Za każdym razem stosowanie niestandardowego rozwiązania musi wynikać z analizy sytuacji, a nie z nieznamomości zasad. W dalszej części rozdziału omówiono dokładnie proponowane przez nas zasady pisania programów.

Zgodnie z tymi zasadami, pliki zawierające kod źródłowy w języku C dzielimy na pliki nagłówkowe i pliki programowe. Pełnią one całkowicie odmienne zadania. Rozszerzeniami ich nazw są odpowiednio .h i .c.

3.2.1. Pliki nagłówkowe

Każdy plik nagłówkowy opisuje jeden plik programowy. Nazwy ich różnią się jedynie rozszerzeniem. Zadaniem pliku nagłówkowego jest dostarczenie kompilatorowi wszystkich tych danych, jakie są wykorzystywane do wywoływania funkcji zdefiniowanych w przypisanym mu pliku programowym, zwanym dalej plikiem związanym. Aby to osiągnąć, w pliku nagłówkowym konieczne jest zapisanie:

- definicji wszystkich stałych stosowanych w zewnętrznych funkcjach pliku związanego, np. kodów błędów
- definicji wszystkich typów wykorzystywanych w zewnętrznych funkcjach pliku związanego

- deklaracji wszystkich zewnętrznych funkcji z pliku związanego; deklaracje powinny być w pełnej postaci, tzn. muszą zawierać - oprócz typu funkcji - także typy parametrów
- deklaracji wszystkich zmiennych zewnętrznych z pliku związanego.

Wprowadzonych pojęć deklaracja i definicja nie można stosować zamiennie. W publikacji przyjmuje się, że deklaracja określa typ obiektu (np. funkcji lub zmiennej), a definicja powoduje także przydzielenie temu obiektowi obszaru pamięci, w jakim jest on przechowywany. Tak więc możliwych jest wiele deklaracji zmiennej, ale tylko jedna jego definicja. Dokładniejsza analiza tych terminów jest zamieszczona w p. 1.1.

Pliki nagłówkowe nie powinny zawierać żadnych innych konstrukcji języka. Umieszczanie w tych plikach np. definicji zmiennych jest błędem. Rozważmy, jakie konsekwencje ma umieszczenie definicji zmiennej z wykorzystywanej w wielu funkcjach pliku programowego o nazwie `a`, w jego pliku nagłówkowym:

```
a.h:    int z;
        ...

a.c:    #include <a.h>
        ...
        a_f1()
        {
            z++;
            ...
        }
        a_f2()
        {
            z--;
            ...
        }
```

Rodzaj operacji wykonywanych na zmiennej `z` jest naturalnie nieistotny, ważne jest natomiast, że zmienna `z` może być użyta we wszystkich funkcjach `a.c`. Tak napisany program jest syntaktycznie poprawny i zostanie poprawnie wykonany. Sytuacja ta ulegnie zmianie, gdy inne pliki programowe, tworzące jeden program, będą chciały skorzystać z funkcji `a.c` i w tym celu zamieszczą dyrektywę preprocesora

```
#include <a.h>
```

dopisującą plik nagłówkowy a.h do ich treści tak, aby skorzystać np. z zamieszczonych tam deklaracji funkcji. Kompilacja każdego z nich przebiegnie poprawnie, lecz nastąpi błąd na etapie łączenia programu. Zmiennej zewnętrznej z zostanie bowiem wielokrotnie przydzielona pamięć. Analogiczny błąd wystąpi podczas wpisywania definicji funkcji do pliku nagłówkowego.

Pominięcie deklaracji funkcji może prowadzić do powstania trudnych do usunięcia błędów wykonania. Brak deklaracji uniemożliwi naturalnie sprawdzanie poprawności przekazywania parametrów. Konsekwencje te mogą być jednak jeszcze większe - gdy brakuje deklaracji, kompilator zakłada, iż funkcja jest typu int. Powoduje to utratę fragmentu danych w sytuacjach, gdy faktycznie zwracana jest wartość innego typu np. long lub wskaźnik.

Definicje stałych, makr i typów powinny być włączone do plików programowych z pliku nagłówkowego. Wpisanie ich bezpośrednio do plików programowych jest poprawne syntaktycznie i pozwala na połączenie programu, ale jakakolwiek ich zmiana powoduje konieczność poprawiania wszystkich plików, w których zostały one zapisane, co jest uciążliwe. Jeżeli któryś z plików nie zostanie poprawiony i/lub skompilowany, to może doprowadzić to do wystąpienia trudnych do wykrycia błędów wykonania, np. dodanie nowego pola p do struktury powoduje błędne naliczanie adresów wszystkich pól wymienionych w definicji po polu p. Umieszczenie definicji typów w pliku nagłówkowym i stosowanie programu **MAKE** pozwala na automatyczne uzyskanie poprawnej wersji programu wynikowego (patrz p. 7.2).

3.2.2. Pliki programowe

Pliki programowe zawierają definicje funkcji i zmiennych. Zwykle plik programowy zawiera funkcje jednej warstwy i jest on pisany przez jednego programistę. Jego wielkość jest uzależniona od zadania spełnianego przez tę warstwę. Należy jednak pamiętać, że poprawianie plików już o rozmiarach kilkunastu KB jest kłopotliwe, nawet za pomocą wyspecjalizowanych edytorów (patrz p. 7.1). W trakcie kompilacji do pliku programowego dołączane (doczytywane) są pliki nagłówkowe. W wyniku otrzymujemy plik o rozszerzeniu nazwy .OBJ, nazywany dalej modulem. Plik programowy składa się z nagłówka, definicji zmiennych i definicji funkcji.

3.2.2.1. Nagłówek pliku programowego

Nagłówek jest to początkowy fragment pliku, pomijany przez kompilator (jest on zapisany jako komentarz), w którym zamieszczamy następujące informacje:

- nazwę pliku i skrótowe opisanie zadania przez niego realizowanego
- wykaz pełnych deklaracji wszystkich funkcji zewnętrznych pliku z krótkim, jednozdaniowym opisem spełnianego przez nie zadania
- wykaz deklaracji zmiennych zewnętrznych z opisem pełnionych przez nie funkcji
- wykaz plików programowych, zawierających wykorzystywane niestandardowe funkcje
- nazwisko autora projektu i kodu
- datę zakończenia prac
- wykaz dokonanych poprawek; każda pozycja powinna zawierać: datę, nazwisko osoby wprowadzającej zmianę, powód jej wprowadzenia, miejsce i opis dokonanych zmian.

Często praktykowane jest pisanie nagłówka pliku na etapie projektowania. Dobrze napisany nagłówek sprawia, że podczas pisania wieloplikowego programu nie zachodzi w zasadzie konieczność oglądania treści funkcji. Wszystkie najważniejsze informacje zgromadzone są w jednym miejscu, co ułatwia ich przeszukiwanie. Zaletą ta stanie się jednak źródłem błędów, gdy programista nie zadba sam o uaktualnienie zawartości nagłówka po dokonaniu poprawek.

3.2.2.2. Typy funkcji

Wśród funkcji wyróżniamy: funkcje zewnętrzne, wewnętrzne i testujące. Funkcje zewnętrzne realizują zadania wykonywane przez warstwę, ich deklaracje są zapisane w odpowiednim pliku nagłówkowym. Funkcje wewnętrzne są funkcjami roboczymi, wprowadzonymi w celu ułatwienia pisania funkcji zewnętrznych. Ich deklaracje są umieszczone w pliku programowym przed definicjami funkcji. Pozwala to kompilatorowi na sprawdzenie poprawności ich wywoływania. Funkcje testujące, wykorzystywane do uruchamiania programu, są omówione w kolejnym podrozdziale.

Przy określaniu rozmiaru kodu definiującego funkcje należy uwzględnić następujące czynniki:

1. Wiele użytecznych funkcji składa się wyłącznie z instrukcji `return` zwracającej określoną wartość. Dzieje się tak wówczas, gdy np. funkcja udostępnia zawartość pola w lokalnej strukturze.
2. Psychologowie stwierdzili, iż liczba obiektów, jakie przeciętny człowiek jest w stanie kontrolować bez wysiłku, mieści się w przedziale od 5 do 9. Zasada ta wyznacza stopień złożoności funkcji. Obiektami są tutaj zmienne, wywołania funkcji, pętle. Jeżeli w pewnym fragmencie funkcji liczba obiektów przekracza 9, to należy zadania realizowane przez kilka obiektów wykonać za pomocą jednej, specjalnie powołanej do tego celu funkcji roboczej.
3. Liczba wierszy funkcji nie powinna przekraczać 20, tak by w czasie uruchamiania cała jej treść była widoczna na ekranie. Liczba ta może być przekroczona, gdy treść funkcji składa się z sekwencyjnego wykonania prostych działań, lub też stosowana jest instrukcja `switch`.

Treści funkcji towarzyszy opcjonalny komentarz opisujący np. trudny do zrozumienia jej fragment lub dokonaną poprawkę. Należy pamiętać, że dobrze napisany komentarz powinien pozwolić na napisanie opisywanego przez niego fragmentu kodu.

3.2.2.3. Testowanie pliku programowego

Po napisaniu funkcji konieczne jest ich przetestowanie. Przygotowanie testowych wywołań funkcji jest związane niejednokrotnie z dużym nakładem pracy. Funkcje pliku powinny być sprawdzone dokładnie poza jakimkolwiek systemem. Nie można bowiem wykluczyć, że poprawka dobra dla systemu X, nie spowoduje błędu w działaniu systemu Y, stosującego poprawiony również plik. Z tego powodu funkcje testowe zawierają funkcję `main`, konieczną do uzyskania wykonywalnego kodu. W funkcjach tych są przygotowane dane i wywoływane są wewnętrzne i zewnętrzne funkcje testowanego pliku. Jeżeli wynik działania funkcji nie jest oczywisty, to celowe jest umieszczenie, po wywołaniu funkcji, komentarza z opisem prawidłowego wyniku.

Po wykonaniu jakichkolwiek poprawek konieczne jest sprawdzenie, czy nie spowodowały one niepożądanych efektów ubocznych. Zestaw testujących funkcji powinien być zatem dostępny przez cały czas użytkowania pliku. Najbardziej eleganckim rozwiązaniem, nie wymagającym ingerencji w kod źródłowy pliku, jest wykorzystanie kompilacji warunkowej (patrz p. 3.2). Funkcje testowe są umieszczane na końcu pliku między dyrektywami preprocesora:

```
#ifdef X
```

```
...
```

```
#endif
```

gdzie X jest identyfikatorem definiowanym tylko podczas testowania pliku. Jest on zwykle podobny do nazwy pliku, często stosuje się identyfikator utworzony z nazwy pliku poprzedzonej ciągiem "test_", np. dla pliku dzd jest nim test_dzd. Identyfikator ten definiuje się przy kompilacji zewnętrznej za pomocą opcji -d kompilatora, a w środowisku zintegrowanym przez wpisanie go w podopcji DEFINES opcji OPTIONS (Borland C++ 2.0).

3.2.2.4. Przykład

Poniżej zamieszczono fragmenty pliku *inwers.c*, napisane zgodnie z wymienionymi zaleceniami. Pełny opis funkcji w nim umieszczonych jest podany w p. 6.3.2.

```

/*****
  INWERS: obsługa tablicy inwersyjnej wersja 1.0
  *****/

Funkcje zewnętrzne:
    int inv_ini(char *nazwa_pl, char *terminy[], ops_inv *inv)
        zakładanie nowej tablicy inw., liste terminow konczy NULL
    int inv_open(char *nazwa_pl, ops_inv *inv)
        otwieranie istniejącej tablicy
    int inv_close(ops_inv *inv)
        zamykanie tablicy
    int inv_get(ops_inv *inv, ops_ter *dane)
        zapelnienie *dane, nazwa terminu w dane->termin przed
        wywołaniem
    ....

Zmienne zewnętrzne:
    brak

Wykorzystywane pliki programowe:
    dzd

*****

Autor: A. Sieminski
data : 1991.9.20
*****

Poprawki:
*****/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <alloc.h>
#include <dir.h>
#include <dzd.h>
#include <inwers.h>

static char licznosc[]="@lt@";
static int por_dane(typ_ter *t1, typ_ter *t2);
static int dzd_ter(ops_ter *o, dzd *d);
static int ter_dzd(dzd *d, ops_ter *o);
static int bada_unsigned(unsigned *l1, unsigned *l2);
static int por_dane(typ_ter *t1, typ_ter *t2)
{
    return( strcmp((*t1).termin, (*t2).termin));
}

...

#ifdef test_inwers
main(int lp, char *par[])
{
    int wyn;
    char *t[] =
    {
        "test", "nie jade", "pojade", "pojedziemy",
        NULL,
    };
    ops_dzd teksty;
    ops_ter ts;
    ops_inv *inv, inww;
    inv=&inww;
    if (lp!=2)
    {
        puts("inwers plik_odwrocony");
        exit(1);
    }
    if (inv_ini(par[2],t,inv)==ERR)
    {
        puts("nie utworzyłem inwers!");
        exit(1);
    }
    ts.termin="nowy";

```

```
ts.dl_danych=0;
wyn=inv_ins(inv,&ts);
wyn=inv_get(inv,&ts);

...
inv_close(inv);
if (inv_ini(par[2],t,inv)==ERR)
{
    puts("nie utworzyłem invers!");
    exit(1);
}
ts.termin=strdup("test");
wyn=inv_get(inv,&ts);
ts.termin=strdup("pojedziemy");
wyn=inv_get(inv,&ts);
inv_close(inv);
}#endif
```

3.3. Optymalizacja

Cechą wyróżniającą profesjonalnego programistę spośród jego domorosłych kolegów jest umiejętność pisania programów o jak najkrótszym czasie wykonania. Programów, które efektywnie wykorzystywałyby dostępne narzędzia, były niezawodne, łatwo przenośne oraz miały przejrzystą strukturę. Jednoczesne spełnienie wszystkich tych wymogów nie jest jednak zwykle możliwe. Najprostszą miarą efektywności programu jest całkowity czas jego działania. Z tego punktu widzenia przyspieszenie działania można na przykład uzyskać przez zastąpienie wywołań funkcji w języku C wstawkami w assemblerze, co jednocześnie zmniejsza czytelność kodu i utrudnia jego modyfikację. Innym kryterium jest zmniejszenie wykorzystania zajmowanego obszaru pamięci dyskowej. W tym celu używa się różnych metod kompresji. Powstaje tu jednak konflikt między szybkością działania programu i wykorzystywaniem pamięci komputera. Nie zawsze jednak mamy do czynienia z tak niekorzystną sytuacją. Kilka z przedstawionych w tym rozdziale metod pozwala na skrócenie kodu wynikowego z jednoczesnym przyspieszeniem czasu jego tworzenia i wykonywania programu.

Zaznając się z treścią poniższego rozdziału należy pamiętać, że w rzeczywistości najlepsze wyniki daje dobór właściwego algorytmu przetwarzania. Optymalizacja już napisanego kodu przyspieszy wykonanie programu z reguły nie więcej niż o kilkanaście procent, natomiast zmiana algorytmu może ten czas skrócić kilkakrotnie (patrz algorytm

obliczający liczby Fibbonaciego w p. 7.6.4). Do identyfikacji fragmentów programu, które należy spróbować zoptymalizować, służy profiler opisany w p. 7.5.

3.3.1. Inkrementacja i dekrementacja

Język C został wyposażony w dwa operatory inkrementacji ++ (preinkrementacja i postinkrementacja) oraz w dwa operatory dekrementacji -- (predekrementacja i postdekrementacja). Funkcjonalnie operator inkrementacji zastępuje następujące wyrażenie:

```
x = x + 1;
```

Podobnie jest z dekrementacją. Oprócz oczywistego zysku podczas zapisu, operatory te są szybsze od ich odpowiedników i używają mniej pamięci RAM niż operatory przypisania. Przyczyna leży w strukturze kodu generowanego przez kompilator. Można to zaobserwować po podaniu w linii zlecenia kompilatora parametru -S. Tworzony jest wtedy wydruk z instrukcjami w assemblerze, wygenerowanymi w trakcie kompilacji programu.

Inkrementacja typu x++ lub ++x jest zamieniana na inc si. Natomiast instrukcja podstawienia x = x + 1; tłumaczona jest w następujący sposób:

```
mov ax,si
inc ax
mov si,ax
```

Wniosek: w pierwszym przypadku kod wynikowy jest krótszy i szybszy.

Mimo tych zalet istnieją sytuacje, w których należy unikać stosowania operatorów inkrementacji i dekrementacji. Należy do nich m.in. stosowanie ich do wywoływania makr i kilkakrotnego przekazywania zmiennych z tymi operatorami. Przykład:

```
#define MAX(x,y) (((x)>(y))?(x):(y))
```

```
...
```

```
int i=6, j=5;
```

```
printf("%d",MAX(i++,j--));
```

spowoduje wypisanie wartości 7. Natomiast efektem niepożądanym będzie nadanie zmiennej i wartości 8, a zmiennej j wartości 4.

Uwaga. Nie zaleca się stosowania tych operatorów do przekazywania parametrów funkcji. Język C nie określa bowiem kolejności obliczania parametrów aktualnych w wywołaniach funkcji.

3.3.2. Zmienne rejestrowe

Istnieją dwie ważne przyczyny, by zawsze stosować zmienne rejestrowe w pętlach, jeżeli jest to tylko możliwe:

1. Zmienna rejestrowa jest tworzona w wewnętrznym rejestrze procesora i czas dostępu do niej jest bardzo krótki - o wiele krótszy niż w przypadku pamięci operacyjnej.
2. Szybkość wykonywania pętli jest wartością krytyczną i rzutuje ona na czas wykonywania całego programu.

Następujący program, skompilowany do postaci assemblerowej, pokazuje różnicę między zmienną rejestrową i zmienną pamięciową.

```
int j;
main()
{
    register int i;
    for (i=0; i<100; i++) ;
    for (j=0; j<100; j++) ;
}
```

Tak wygląda interesujący nas fragment kodu z komentarzem:

```
;    register int i;
;
;    for (i=0; i<100; i++);
;
    xor si,si          ; początek pętli
    jmp short @1@74
@1@50:
    inc si              ; inkrementacja zmiennej rejestrowej
@1@74:
    cmp si,100          ; wymagane jest tylko porównanie rejestrowe
    jl short @1@50
;
;    for (j=0; j<100; j++);
;
    mov word ptr DGROUP:_j,0 ; początek pętli
    jmp short @1@146
@1@122:
    inc word ptr DGROUP:_j   ; inkrementacja zmiennej pamięciowej
@1@146:
    cmp word ptr DGROUP:_j,100 ; porównanie wymaga dostępu do pamięci
    jl short @1@122
```

Jak wielka jest ta różnica, widać dopiero po dokładnym porównaniu czasów wykonania. Przyjmijmy, że program jest wykonywany na procesorze 8088. W pętli używającej zmiennej rejestrowej instrukcja `inc si` inkrementuje zmienną kontrolną. To wymaga tylko dwóch taktów zegarowych. Z drugiej strony, pętla ze zmienną pamięciową używa instrukcji `inc word ptr DGROUP:_j`, która wymaga 29 taktów. Różnica jest prawie piętnastokrotna!

Kompilator Borlanda automatycznie przekształca typ dwóch pierwszych zmiennych całkowitych, zdefiniowanych w funkcji na typ rejestrowy, jeśli tylko żadne inne zmienne rejestrowe nie są używane. Ten proces zmniejsza całkowity czas wykonania programu. Zalecamy jednak stosowanie słowa kluczowego **register** do oznaczenia zmiennych rejestrowych i przemyślenie kolejności definiowania zmiennych. Zmienne rejestrowe powinny być stosowane oczywiście w najbardziej zagnieżdżonych pętlach.

Uwaga. Zmienne globalne nie mogą być umieszczane w rejestrach. Nie można stosować operatora `&` do zmiennych rejestrowych.

3.3.3. Wskaźniki i tablice

Do elementów tablic można się odwoływać w języku C na dwa sposoby: przez wyrażenia indeksowe (jedyne dostępne sposoby np. w języku Pascal) oraz przez wskaźniki. Drugi sposób jest efektywniejszy (kompilator automatycznie tłumaczy wyrażenia indeksowe na wskaźnikowe).

Przyjrzyjmy się dwóm fragmentom kodu, wykonującym te same działania:

```
char array[100], *ptr;
int i;
for (;;)
{
    a = array[i++];
    ...
}

ptr=array+i;
for (;;)
{
    a = *(ptr++);
    ...
}
```

Na korzyść metody wskaźnikowej przemawia fakt, że do pobrania kolejnego elementu tablicy potrzebna jest tylko inkrementacja adresu. W drugim przypadku, oprócz inkrementacji zmiennej indeksowej, za każdym wykonaniem pętli konieczne jest także uwzględnienie adresu początku tablicy. Różnica w działaniu jest jeszcze bardziej widoczna podczas używania tablic wielowymiarowych. Największe korzyści ze stosowania wskaźników osiąga się podczas sekwencyjnego przeglądania tablicy, np. poszukiwania minimalnego elementu wielowymiarowej tablicy liczb.

```
int a[8][3][70];
int i, j, k, min;
int *pocz, *kon;

/* metoda indeksowa */
for(min=MAXINT, i=0; i<8; ++i)
    for (j=0; j<3; j++)
        for (k=0; k<70; ++k)
            min = (a[i][j][k]<min)? a[i][j][k]: min;

/* metoda wskaźnikowa */
min=MAXINT, pocz=&a[0][0][0], kon=pocz+sizeof(a);
for(; pocz<=kon; pocz++)
    min = (*pocz<max)? *pocz : min;
```

Podczas przetwarzania 'wyrywkowego' wyrażenia wskaźnikowe są mało czytelne, co może łatwo doprowadzić do popełnienia błędu. Ilustruje to poniższy przykład:

```
int a[2][3][2] =
    ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 );
```

Przeanalizujemy tabelę, w której pierwsza kolumna ilustruje notację wskaźnikową do elementów, druga odpowiadające wyrażenia indeksowe, trzecia wartość tych wyrażeń:

wyr. wskaźnikowe	wyr. indeksowe	wartość
*a	a[0]	{{0,1},{2,3},{4,5}}
*(a+1)	a[1]	{{6,7},{8,9},{10,11}}
**a	a[0][0]	{0,1}
*(a+1)	a[0][1]	{2,3}
*(a+3)	a[1][0]	{6,7}
***a	a[0][0][0]	0
*(**a+1)	a[0][0][1]	1
*(**a+6)	a[1][0][0]	6

Sposób dokonywania translacji wyrażeń indeksowych, odnoszących się do tablicy `tab[A][B][C][D]` na wyrażenia wskaźnikowe, definiuje poniższy przykład:

1. `tab[a]` = `*(tab + a)`,
2. `tab[a][b]` = `*(tab + B*a + b)`,
3. `tab[a][b][c]` = `*(tab + C*B*a + C*b + c)`,
4. `tab[a][b][c][d]` = `*(tab + D*C*B*a + D*C*b + D*c + d)`, itd.

W powyższym przykładzie 11 element tablicy `a` można uzyskać na kilka sposobów, całkowicie niedostępnych w standardowym widzeniu tablicy:

`a[0][0][11]=a[0][1][9]=a[0][2][7]=...=a[1][0][5]=11.`

Wynika to stąd, że równanie $2*3*a + 2*b + c = 11$ ma aż 9 rozwiązań, m.in. te podane wyżej.

Jako ćwiczenie sprawdzające zrozumienie takiego podejścia do pobierania elementów tablic proponujemy zastanowienie się, czy poniższe wyrażenia są równoważne:

```
int a[2][3][2] =
*(a+3)          a[1][0]          **(a+1)
**a+6           a[1][0][0]       ***a+1
```

Używanie wskaźników do obsługi tablic kryje jednak w sobie wiele niebezpieczeństw. Nie każde wyrażenie wskaźnikowe daje się zamienić poprawnie na odpowiadającą mu postać indeksową. Zastanówmy się nad postacią `*(a+1)+i`, gdzie $i = 0, 1 \dots 9$. Odpowiednio:

```
*(a+1)+0        daje jako rezultat   {2,3}
*(a+1)+1        daje jako rezultat   {3,4}
*(a+1)+2        daje jako rezultat   {4,5}
.....
*(a+1)+9        daje jako rezultat   {11,0}
```

W tym wypadku jedynie parzyste wartości `'i'` dają poprawny rezultat, tj. zgodny ze sposobem zdefiniowania tablicy `a`. Dla wartości `'i'` nieparzystych otrzymujemy parę liczb leżącą na granicy zdefiniowanych par, nieosiągalną dla wyrażeń indeksowych. Taki nieprawidłowy sposób odwoływania się do tablicy jest różnie interpretowany przez oprogramowanie narzędziowe, np. debugger środowiska zintegrowanego Borland C++ 2.0 udostępnia nieprawidłowo określone tablice, co nie występuje podczas stosowania Turbo Debuggera 2.5, który poprawnie interpretuje wskaźnik na liczbę typu `int`. W

zdecydowanej większości przypadków (oprócz już opisanych) różnica w efektywności wygenerowanego kodu jest minimalna i znacznie bezpieczniej jest zaufać kompilatorowi i stosować wyrażenia indeksowe.

3.3.4. Wywoływanie funkcji

Po wywołaniu funkcji na stos są odkładane: wszystkie lokalne zmienne, argumenty wywoływanej funkcji i adres powrotu do funkcji wywołującej. Kiedy funkcja kończy swoje działanie, adres ten, wszystkie argumenty i lokalne zmienne są usuwane ze stosu. Te dwa procesy zabierają dość dużo czasu i niekiedy opłaca się unikać wywołań funkcji, zwłaszcza wtedy, gdy czas jej wykonania niewiele się różni od czasu wymienionych czynności organizacyjnych. Wynikające stąd pogorszenie czytelności programu można zmniejszyć przez stosowanie makr. Należy pamiętać, że powoduje to zwykle wzrost wielkości kodu wynikowego.

Jest to klasyczny przykład potwierdzający zasadę, iż zmniejszenie kodu wynikowego dobrze napisanego programu można osiągnąć tylko kosztem szybkości wykonania i na odwrót, zwiększenie szybkości powoduje rozrost kodu.

3.3.5. Prekompilowane pliki nagłówkowe

O optymalizacji można mówić dwójako: o optymalizacji kodu, jak i sposobie jego tworzenia. W ramach drugiej możliwości stosuje się tzw. prekompilowane pliki nagłówkowe (headery).

Podczas kompilacji wielomodułowych programów w języku C kompilator może zaoszczędzić nawet połowę czasu przetwarzania (parsingu) plików nagłówkowych. Sama analiza polega na zapisywaniu do specjalnej tablicy symboli wszystkich deklaracji i definicji napotkanych w tych plikach. Jeśli np. dziesięć plików źródłowych włącza ten sam plik nagłówkowy, to ten plik podlega przetwarzaniu dziesięć razy, za każdym razem produkując tę samą tablicę symboli. Prekompilowane pliki nagłówkowe skracają ten proces. Podczas pierwszej kompilacji danego pliku nagłówkowego kompilator tworzy kopię tablicy symboli na dysku. Później, jeśli tylko nastąpi próba kompilacji tego samego zbioru, kompilator ładuje tablicę symboli danymi z tego pliku zamiast dokonywać jego ponownej analizy. Bezpośrednie załadowanie tablicy z dysku jest ponad dziesięć razy szybsze niż kolejne przetwarzanie tego samego zbioru.

Prekompilowane pliki nagłówkowe są używane tylko wtedy, gdy w tablicy zapisano ich aktualną wersję, a kompilacja jest wykonywana z tymi samymi parametrami.

Prekompilację plików nagłówkowych z zastosowaniem kompilatora Borland C++ 2.0 można uzyskać na trzy sposoby:

1. W zintegrowanym środowisku używając sekwencji Alt O C C.
2. Podając w linii zlecenia kompilatora `bcc -H, -Hu` lub `-H=nazwa_pliku`.
3. Programowo przez `pragma hdrfile` i `hdrstop`.

3.3.6. Inkrementalna kompilacja i łączenie

Inkrementalna kompilacja i łączenie pomagają przyspieszyć proces tworzenia wykonywalnego programu. Ogólnie proces ten polega na wymianie tylko zmienionych fragmentów kodu. I tak w przypadku inkrementalnej kompilacji kompilowane są tylko zmodyfikowane funkcje, w przypadku inkrementalnego łączenia zamieniane są w kodzie wynikowym tylko moduły zawierające te zmienione funkcje. Opcja ta jest dostępna w kompilatorze Microsoft C 6.0 i kontrolują ją następujące opcje:

`/Gi` - wybrana z `/qc` powoduje inkrementalną kompilację, kompilowane są tylko funkcje zmienione w stosunku do poprzedniej kompilacji. Jednocześnie automatycznie wywołuje się ILINK - inkrementalny program łączący (linker). Zastosowanie `/Gi` bez podania `/qc` nie spowoduje kompilacji inkrementalnej, ale umożliwia stosowanie inkrementalnego łączenia.

`/Li[liczba]` - łączy inkrementalnie wywołując ILINK. Opcjonalnie podawana liczba określa wielkość rekordów stosowanych podczas łączenia. Dokładniejsze informacje na ten temat można znaleźć w pracy [MC].

Programu ILINK powinno używać się do aktualizacji kodu wynikowego, kiedy część modułów wchodzących w projekt systemu została zmieniona. Modułem w tym przypadku jest pojedynczy plik typu object (.OBJ). W przypadku wystąpienia błędu należy przeprowadzić pełne łączenie, np. `ILINK /E "LINK /INC @A.LNK" A`.

Uwaga. Stosowanie nakładek wyklucza używanie programu ILINK.

3.4. Przenośność oprogramowania

Bardzo często zdarza się, że program raz napisany musi zostać uruchomiony na innym komputerze, z innym systemem operacyjnym.

Adaptacja programu może być w jednym przypadku bardzo prosta, a w innych wręcz niemożliwa do wykonania i wymagać napisania programu na nowo. Język C jest obecnie dostępny na wszystkich rodzajach komputerów, począwszy od mikrokomputerów, skończywszy na superkomputerach. Co więcej, wszyscy producenci kompilatorów dostosowali je do wymogów opisanych w pracy [KR], np. kompilator BORLAND C++ 2.0 opcja -AK. Trend ten uległ wzmocnieniu po zatwierdzeniu normy ANSI C [ANSI], np. kompilator Microsoft C 6.0 opcja /Za. Programy w języku C są znacznie łatwiej przenośne także na inne płaszczyzny systemowe, np. opcja -AU kompilatora BORLAND C++ 2.0, niż np. w języku Pascal, bowiem norma tego języka zaproponowana przez Wirtha [Wirth] jest tak uboga, że nie pozwala na wykorzystanie zgodnych z nią kompilatorów do pisania systemów użytkowych. Sytuacja ta doprowadziła do niekontrolowanego rozwoju języka, czego przejawem jest np. Turbo Pascal.

W tym rozdziale przedstawione są rozwiązania najczęściej spotykanych problemów, związanych z zapewnieniem przenośności programów napisanych w języku C.

3.4.1. Uzależnienia od systemu operacyjnego

Najprostszą drogą do osiągnięcia przenośności programów jest stosowanie dyrektywy `#define` dla każdej stałej zależnej od systemu czy procesora. Liczbą tą może być wielkość bufora dla operacji dyskowych, definicja rozmiaru ekranu czy komend klawiatury – wszystko to, co ma choć najmniejszą szansę, by ulec zmianie po przeniesieniu na inny komputer. Przy takim podejściu unikamy konieczności ingerencji w kod źródłowy funkcji, wystarczy poprawienie pliku nagłówkowego. Zmiana w tym jednym miejscu pozwoli na wygenerowanie zmodyfikowanej wersji kodu. W rzeczywistości, w celu zoptymalizowania czasu działania, producenci profesjonalnego oprogramowania często korzystają z metod programowania nie dających przenieść się na inne komputery lub systemy operacyjne. Tak jest z obsługą ekranu na IBM PC. Kiedy już nie ma innej możliwości, jak zdefiniowanie uzależnionej sprzętowo funkcji, należy to tak przeprowadzić, aby w prosty sposób można to było zaadoptować na innym komputerze, pozostawiając pozostałe funkcje nietknięte, np. funkcja czyszcząca ekran może wyglądać tak:

```
void czyszc_zawsze(char op, x, y)
{
    switch(op)
    {
        case 1: clrscr(); break;
```



```

    case 2: czysc_biose(x,y); break;
  }
}

```

W ten sposób tylko implementacja danej funkcji może ulec w razie potrzeby zmianie, pozostawiając zewnętrzny szkielet nietknięty.

Zasada ta w pełni jest wykorzystana w zalecanym w tej pracy programowaniu wielowarstwowym. Najniższa warstwa jest odpowiedzialna za wszystkie operacje niskiego poziomu (najczęściej nieprzenośne) i zapewnia interfejs z systemem operacyjnym. Podczas przenoszenia kodu można wtedy cały wysiłek skoncentrować na analizie jednego modułu, a nie całego programu.

3.4.2. Różnice w wielkości standardowych danych

Jak wiadomo, w procesorach 16-bitowych wielkość słowa wynosi 16 bitów, a w procesorach 32-bitowych - 32 bity. Ponieważ liczba bajtów zajmowanych przez liczby typu `int` jest taka sama jak wielkość słowa w procesorze, należy być bardzo ostrożnym i nigdy nie zakładać określonej wielkości danych. Dlatego też konieczne jest używanie dyrektywy `sizeof`, gdy program potrzebuje informacji o liczbie zajmowanych bajtów, np. funkcja `write_int` zapisuje zmienną typu `int` do pliku i działa na każdym komputerze poprawnie.

```

void write_int(int i)
{
    fwrite (&i, sizeof(int), 1, stream);
}

```

Czasami jednak takie postępowanie jest nieopłacalne, gdyż koszt poniesiony podczas pisania takiej przenośnej funkcji jest za duży. Jako przykład niech posłuży funkcja `swap`, zamieniająca skrajne bajty. Na procesorze 8088 zmienne typu `int` są 2-bajtowe, natomiast na Motoroli 68000 są 4-bajtowe. Analizę pozostawiamy Czytelnikowi.

```

swap(int *x)
{
    union {
        int t;
        unsigned char c[sizeof(int)];
    } swap;
    unsigned char temp,i;
    swap.t=*x;
    for (i=0; i<(sizeof(int))/2; i++)
    {

```

```
        temp=swap.c[sizeof(int)-1-i];  
        swap.c[sizeof(int)-1-i]=swap.c[i];  
        swap.c[i]=temp;  
    }  
    *x=swap.t;  
}
```

Rozdział 4

EKRAN

4.1. Wprowadzenie

Rozdział ten omawia pakiet funkcji przeznaczonych do obsługi ekranu. Zgodnie z zaleceniami sformułowanymi w rozdz. 3, ekran obsługiwany jest za pomocą 4 warstw: ekranu fizycznego, okien, ekranów wirtualnych i funkcji użytkowych. Kolejne warstwy zapewniają coraz to większy komfort pracy programiście; cena, jaką się za to płaci, to zwiększenie wielkości kodu wynikowego i nieznaczne wydłużenie czasu pracy programu.

W zdecydowanej większości przypadków stosowane są funkcje z dwu najwyższych warstw. Nie oznacza to jednak, że wystarczające jest zaznajomienie się z końcową częścią rozdziału. Znajomość wszystkich warstw pozwala na wybór tej z nich, która jest najwłaściwsza w konkretnej sytuacji. Ponadto wykonanie pewnych operacji jest możliwe tylko za pomocą funkcji niższej warstwy. Programista musi znać nie tylko dostępne w każdej warstwie funkcje, ale też wiedzieć, które z nich można – a nawet trzeba – wykorzystać w wyższych warstwach. W pewnych przypadkach bowiem stosowanie funkcji z niższej warstwy jest konieczne, w innych zaś może to doprowadzić do zaburzeń w pracy funkcji wyższej warstwy. Podkreślić należy jednak, że w wypadku stosowania omawianych poniżej funkcji zdecydowanie odradzamy posługiwanie się standardowymi funkcjami do zmiany zawartości ekranu. Korzyści, jakie można z tego osiągnąć, są minimalne, a prowadzi to niemal zawsze do powstania uprzednio nieprzewidzianych skutków ubocznych.

Rozdział ten można potraktować jak obszerny przykład ilustrujący zasady tworzenia oprogramowania wielowarstwowego.

Zamieszczony tu materiał został opracowany w głównej mierze na podstawie pracy [ADV]. Ekran obsługiwany jest w trybie znakowym. Może to wywołać pewne zdziwienie, obserwuje się bowiem ostatnio rosnącą popularność trybu graficznego. Przyjęte rozwiązanie ma jednak swoje

zalety: bardzo duża szybkość pracy w trybie znakowym rekompensuje jego nieco mniej wyszukany, w porównaniu z trybem graficznym, wygląd ekranu. Zebrane doświadczenia wskazują, że mniejsza liczba środków jakimi dysponujemy w trybie tekstowym, ułatwia zaprojektowanie estetycznie wyglądających ekranów. Najważniejsza zaleta to łatwość uzyskania przenośności kodu źródłowego. Funkcje tu bowiem opisane mogą być wykorzystane nie tylko na mikrokomputerach innych typów, ale również terminalach podłączonych do systemów wielodostępnych. Terminale te pracują zwykle jedynie w trybie tekstowym, a nawet jeśli są one w stanie przejść w tryb pracy graficznej, to praktycznie nie można z tej możliwości korzystać z powodu niewielkich szybkości transmisji.

4.2. Implementacja funkcji obsługi ekranu

Implementowanie pakietu na potrzeby konkretnego typu komputera, czy też terminala, ogranicza się do adaptacji funkcji z poziomu ekranu fizycznego.

4.2.1. Wymagania ogólne

Przed przystąpieniem do omawiania implementacji konieczne jest określenie wymagań, jakie musi spełnić monitor lub terminal. Nie są one wygórowane. Zakłada się, że ekran ma kształt prostokątny i jest on w stanie wyświetlać kursor oraz znaki, stosując jeden z dostępnych atrybutów.

Znak jest zapisany na 8 bitach, przy czym znaki o kodach od 32 do 127 są wyświetlane na ekranie w standardzie ASCII. Znaki o kodach powyżej 127 są też dopuszczane do stosowania, można tam znaleźć np. zestaw znaków wykorzystywanych do rysowania ramek. Trzeba pamiętać, że korzystanie z nich jest jednak utrudnione. Brak jest ogólnie przyjętej normy ich kodowania, a co więcej - w wielu sieciach przesyłanych jest tylko 7 mniej znaczących bitów, co sprawia, że są one w ogóle niedostępne. Zastosowane rozwiązanie polega na zdefiniowaniu w headerze `display.h` stałych reprezentujących kody wszystkich znaków wykorzystywanych do rysowania ramek. Nazwy wraz z przypisanymi im wartościami są podane w załączniku.

Jeżeli znaki do rysowania ramek nie są dostępne, to należy je zastąpić innymi znakami z podstawowego zestawu o podobnym wyglądzie, np. znak `C_HD` przez `=`. Moduł obsługi okien zakłada, że mogą one mieć dwa rodzaje ramek: pojedynczą i podwójną. Jeśli terminal nie dysponuje odpowiednimi znakami, to często stosuje się ramkę utworzoną z

zapisanych w inwersie spacji. Oprócz znaków do rysowania ramek w headerze umieszczono też znak `C_HATCH`, służący do wypełniania dużych obszarów.

Jak już wspomniano, zakłada się, że każdy ze znaków może być wyświetlony na jeden z kilku możliwych sposobów. Sposób przedstawienia znaku określa jego atrybut. Stosowanie atrybutów zwiększa czytelność ekranu, ale jednocześnie sprawia wiele kłopotów z uzyskaniem przenośności oprogramowania. Stopień normalizacji jest tu jeszcze mniejszy niż w przypadku znaków wykorzystywanych do rysowania ramek. Ogólnie zaleca się bardzo oszczędne stosowanie atrybutów, ponieważ:

- ułatwia to przeniesienie aplikacji do środowiska, w którym dysponujemy mniejszą liczbą atrybutów
- pozwala na uzyskanie dobrego wrażenia estetycznego, użytkownik nie jest zaskakiwany nieprzewidzianymi, źle wyglądającymi zestawieniami atrybutów
- w większości przypadków użytkownicy nie zwracają uwagi na "subtelne" zmiany wyglądu ekranu.

Liczba stosowanych atrybutów wynosi 5. Zostały one dobrane tak, by wykorzystać możliwości monitorów monochromatycznych. Plik nagłówkowy `display.h` zawiera definicje 5 atrybutów `A_NORM`, `A_INVERSE`, `A_INTENCE`, `A_BLINK`, `A_UNDER`. Sposób wykorzystania atrybutów dla monitorów monochromatycznych i kolorowych jest opisany w p. 4.6.

Jak już wspomniano, z każdym monitorem związany jest kursor. Zakłada się, że dysponujemy możliwościami programowego przemieszczania go po ekranie, zmiany jego rozmiarów, a także całkowitego usunięcia go z ekranu. Nie zakłada się natomiast, że można ustalić położenie kursora. Jeżeli jest to konieczne, to programista musi sam zadbać o zapamiętanie w zmiennych położenia kursora.

4.2.2. Podstawowe makra i funkcje dla operacji na ekranie

Definicje podstawowych funkcji i makr stosowanych podczas obsługi ekranu są zamieszczone w pliku nagłówkowym `display.h`. Oprócz wspomnianych już uprzednio definicji znaków tworzących ramki, kodów atrybutów, a także kodów kierunków przesuwania (patrz p. 4.3) oraz rodzajów ramek (patrz p. 4.4) są tam zdefiniowane stałe:

```
#define ROWS 24      /* liczba wierszy ekranu fizycznego */
#define COLS 80      /* liczba kolumn ekranu fizycznego */
#define GIANT 9999
```

Wielkość stałej GIANT musi przekraczać rozmiar największego ekranu fizycznego, czy też wirtualnego (patrz p. 4.5). Stosuje się ją w celu zaznaczenia, że dana operacja ma być wykonana do końca ekranu, niezależnie od jego wielkości.

Ekran fizyczny, okna i ekrany wirtualne składają się z prostokątnych obszarów utworzonych ze znaków, z których każdy ma określony atrybut. Każdy znak jest opisywany strukturą **CELL**, zdefiniowana następująco:

```
typedef struct s_cell
{
    char chr;
    int att;
} CELL;
```

Prostokątne obszary na ekranie są określone przez strukturę **RECT**:

```
typedef struct s_rect
{
    short r1, c1;    /* róg górny lewy */
    short r2, c2;    /* róg dolny prawy */
} RECT;
```

Wiersze i kolumny są numerowane począwszy od 0. Funkcje modułu wykonują wiele operacji na prostokątach. Najczęściej stosowane z nich są realizowane za pomocą makr: **RWIDTH** i **RHEIGHT**. Podają one odpowiednio liczbę kolumn i wierszy prostokąta. Oprócz nich stosuje się także makro **RCPY**, służące do kopiowania opisów prostokątów oraz makro **RASG** do nadawania im wartości:

```
#define RWIDTH(a) ((a)->c2 - (a)->c1 + 1)
#define RHEIGHT(a) ((a)->r2 - (a)->r1 + 1)
#define RASG(a,row1,col1,row2,col2) ((a)->r1=row1,(a)->c1=col1,\
    (a)->r2=row2, (a)->c2=col2)
#define RCPY(dst,src) ((dst)->r1=(src)->r1, (dst)->c1=(src)->c1,\
    (dst)->r2=(src)->r2, (dst)->c2=(src)->c2)
```

Chociaż operacje te są wykonywane przez makra, zakłada się, że ich parametry są wskaźnikami. Pozwala to na wykorzystywanie ich do przetwarzania parametrów funkcji. Oprócz tego header ten zawiera często spotykane definicje i makra:

```
#define FALSE 0
#define TRUE 1
#define MIN(x,y) ((x)<(y)? (x):(y))
#define MAX(x,y) ((x)>(y)? (x):(y))
typedef int BOOLEAN;
```

oraz deklarację struktury *GIANT_RECT*

```
RECT GIANT_RECT = {0, 0, GIANT, GIANT};
```

stosowanej często do wypełniania ekranów i okien.

W celu zwiększenia niezawodności oprogramowania, przed wykonaniem jakiegokolwiek operacji zmieniającej zawartość ekranu zasięg zmian jest ograniczany do faktycznych rozmiarów ekranu czy też okna. Przydatna jest do tego celu funkcja *intersect*.

```
BOOLEAN intersect (RECT *srcp, int maxrow, int maxcol, RECT *dstp)
{
    dstp->r1=MAX(0,srcp->r1);
    dstp->c1=MAX(0,srcp->c1);
    dstp->r2=MIN(maxrow,srcp->r2);
    dstp->c2=MIN(maxcol,srcp->c2);
    return((BOOLEAN)(dstp->r1<=dstp->r2 && dstp->c1<=dstp->c2));
}
```

Jak widać **dstp* jest prostokątem powstałym po ograniczeniu rozmiarów prostokąta **srcp* do rozmiarów ograniczonych przez wartości parametrów *maxrow* i *maxcol*. Funkcja ta jest umieszczona w pliku *dsputil.c*. Są też tam zapisane funkcje służące do wykonywania operacji na obszarach pamięci, jakie są niedostępne w niektórych implementacjach języka C: *repmem* (wypełnienie obszaru podanym wzorcem o dowolnej długości zadana liczbę razy), *xmalloc* (alokowanie obszaru pamięci i wypełnianie jej zerem).

4.2.3. Implementacja *IBM_mem*

Biblioteka *CTLS* zawiera implementację funkcji obsługi ekranu o nazwie *IBM_mem*. Została ona opracowana dla IBM/PC AT, pracującego pod kontrolą systemu operacyjnego DOS. W implementacji tej stosuje się wpisywanie danych w obszar pamięci ekranu. Zamieszczone poniżej omówienie *IBM_mem* może być przydatne podczas samodzielnego pisania własnych implementacji. Dokładniejsze informacje na ten temat, wraz z implementacją dla systemu DOS, wykorzystującą do wyświetlania znaków BIOS wraz z trzema implementacjami dla UNIX-a, można znaleźć w pracy [ADV].

W implementacji *IBM_mem* przyjęto, że wszystkie operacje przeznaczone do zmiany zawartości ekranu fizycznego nie powodują wpisania danych bezpośrednio do pamięci ekranu, lecz jedynie do tablicy *dspbuf*:

```
#define MAXROW 24
#define MAXCOL 79
static union
{
    CELL cell;
    short word;
} dspbuf[MAXROW+1][MAXCOL+1];
```

Dspbuf zdefiniowano jako unię, a nie jako strukturę, bowiem czasami wygodniej odwoływać się do jej elementów z uwzględnieniem ich podziału na znak i atrybut (np. podczas wpisywania ciągu znaków), a w innych przypadkach (np. przesuwanie zawartości prostokąta) podział ten jest nieistotny i działanie szybciej zostanie wykonane, gdy będziemy się odwoływać do tych dwu bajtów jak do jednej liczby – nie wszystkie kompilatory języka C zezwalają, bowiem na przypisywanie wartości strukturom.

Uaktualnienie zawartości ekranu następuje dopiero po wykonaniu funkcji *PSsynch* (patrz p. 4.3.1). Oznacza to, że najpierw projektujemy ekran za pomocą funkcji wypisujących dane, a następnie jednocześnie wyświetlamy na ekranie wszystkie dokonane zmiany. Zwiększa to subiektywną szybkość pracy programu, pozwala na uniknięcie pewnych niekorzystnych zjawisk, jak np. migotanie ekranu podczas jego czyszczenia oraz zmniejsza liczbę funkcji bezpośrednio kontaktujących się ze sprzętem. W omawianej implementacji służy do tego tylko jedna funkcja. Dzięki temu można ją napisać w sposób bardzo staranny i poświęcić czas na jej optymalizację. Praktycznie byłoby to nie do wykonania w sytuacji, gdyby takich funkcji było wiele.

Funkcja wpisująca dane do obszaru pamięci ekranu jest funkcja *WRTSCRN*. W *PS_MM.C* jest ona zdefiniowana następująco:

```
void WRTSCRN (dspbuf, pocz, dl)
char *dspbuf; /* wskaźnik na początek bufora dspbuf */
int pocz;     /* liczba początkowych bajtów do pominięcia */
int dl;       /* liczba unii CELL do przesłania */
{
    int start, length;
    int w_pocz, k_pocz;
    int w_kon, k_kon;
    start=pocz>>1;
    length=dl;
    w_pocz=start/COLS;
    k_pocz=start - w_pocz*COLS;
    w_kon=(start+length)/COLS;
```



```

k_kon=(start+length)-w_kon*COLS-1;
k_pocz++, k_kon++, w_pocz++, w_kon++;
if (w_kon==w_pocz)
    puttext(k_pocz,w_pocz,k_kon,w_kon,dspbuf+pocz);
else
    (
        int pocz_ost;
        pocz_ost=start+length-k_kon;
        puttext(k_pocz,w_pocz,COLS,w_pocz,dspbuf+pocz);
        puttext(1,w_kon,k_kon,w_kon,dspbuf+(pocz_ost<<1));
        if (w_pocz+1<w_kon)
            puttext(1,w_pocz+1,COLS,w_kon-1,
                dspbuf+pocz+((COLS-k_pocz+1)<<1));
    )
}

```

Zgodnie z przyjętymi założeniami podczas realizacji funkcji **WRTSCRN** skorzystano ze standardowej dla Borland C funkcji **puttext**. Początkowy fragment funkcji to ustalenie (w konwencji Borland C) współrzędnych pierwszego i ostatniego znaku do transmisji. Samo przesłanie danych na ekran odbywa się za pomocą: jednego wywołania **puttext**, jeżeli wiersz początkowy jest zarazem wierszem końcowym, dwu wywołań, jeśli wiersze te sąsiadują ze sobą i trzech - we wszystkich pozostałych przypadkach. Funkcja ta jest wykonywana poprawnie dlatego, że kolejność umieszczania znaku i atrybutu w strukturze **CELL** jest taka sama jak przyjęta podczas organizacji pamięci ekranu.

Ostatnie dwie funkcje implementacji **IBM_mem** służą do obsługi kursora. Są nimi:

```

void BVsetctype(start,end)
int start;    /* początkowy wiersz postaci kursora */
int end;      /* jak wyżej wiersz końcowy */

oraz

void BVsetcpos(row,col)
int row;      /* wiersz w jakim ma być ustawiony kursor */
int col;      /* jak wyżej dla kolumny */

```

Podczas kodowania tych funkcji wykorzystano dobrze znane przerwania systemu BIOS.

4.3. Ekran fizyczny

Ekran fizyczny to najniższa warstwa obsługi ekranu monitora lub terminala. Oprogramowanie tej warstwy zakłada jedynie, iż:

- ekran fizyczny nie może przekraczać rozmiarami faktycznego ekranu, jakim dysponuje użytkownik, może być natomiast od niego mniejszy
- każdy z użytkowników posiada na własność jeden taki ekran
- rozmiary ekranu nie zmieniają się w trakcie pracy programu.

Warstwa ta zawiera zaskakująco niewiele funkcji. Są one jednak wystarczająco wszechstronne, by pozwolić na oprogramowanie wszystkich następnych warstw. Niewielka liczba funkcji pozwoliła na staranną optymalizację, a także ułatwiła przenośność kodu. Jest to bowiem jedyna warstwa podlegająca zmianom podczas adaptacji oprogramowania na potrzeby np. nowego typu terminala.

4.3.1. Omówienie funkcji

Deklaracje poniższych funkcji są zamieszczone w pliku nagłówkowym `pSCREEN.H`. Korzystanie z nich musi rozpocząć się od wywołania funkcji `PSbegin`.

```
void PSbegin() /* inicjowanie pracy warstwy ekranu fizycznego */
```

Zadania wykonywane przez tę funkcję różnią się w zależności od implementacji i mogą obejmować np. przejście w odpowiedni tryb pracy monitora lub terminala. Zawsze jednak wymagane jest wypełnienie spacjami całego ekranu i usunięcie z niego kursora. Koniec stosowania obsługi ekranu sygnalizujemy wywołaniem `PSend`.

```
void PSend() /* zakończenie pracy z warstwą ekranu fizycznego */
```

Wspomniana już poprzednio funkcja `PSsynch` służy do uaktualnienia postaci ekranu zgodnie z zawartością `dspbuf`.

```
void PSSynch() /* synchronizacja zawartości bufora i ekranu */
```

Rozmiary ekranu fizycznego mogą się różnić w zależności od implementacji. Do ich ustalenia służą dwie funkcje `PSheight` i `PSwidth`.

```
int PSheight() /* liczba wierszy ekranu fizycznego */
```

```
int PSwidth() /* liczba kolumn ekranu fizycznego */
```

Wprowadzanie podanych funkcji, w sytuacji gdy zdefiniowano uprzednio odpowiednie stałe, może się wydać nieuzasadnione. Przyjęcie takiego rozwiązania pozwala nam jednakże na pisanie implementacji, w których rozmiar ekranu jest określany dopiero w czasie jego wykonania

i tym samym możliwe jest wykorzystywanie tej samej wersji oprogramowania w sieci mającej terminale o różnych wielkościach ekranu.

Wypisanie ciągu znaków na ekranie uzyskujemy za pomocą funkcji **PSwrite()**.

```
void PWrite(row, col, ncols, str, att) /* wypisywanie ciągu znaków */
int row; /* zmieniany wiersz ekranu */
int col; /* pierwsza zmieniana kolumna */
int ncols; /* maksymalna liczba znaków do wpisania */
char *str; /* zapisywany ciąg */
int att; /* atrybut zapisywanych znaków */
```

Podczas stosowania tej funkcji należy zwrócić uwagę na jej następujące własności:

- miejsce wypisania ciągu jest podane w parametrach funkcji i jest ono niezależne od położenia kursora
- zmianę atrybutów już zapisanego ciągu można uzyskać jedynie przez powtórne jego zapisanie ze zmienionym atrybutem
- polecenia zapisu w nieistniejącej kolumnie i/lub wierszu są ignorowane, zapis odbywa się tylko w jednym wierszu, po dojściu do końca ekranu nie wypisane znaki ciągu są ignorowane, nie następuje przeniesienie ich do kolejnego wiersza
- jeżeli pozwalają na to rozmiary ekranu, to liczba wypisanych znaków wynosi $\text{MIN}(\text{ncols}, \text{strlen}(\text{str}))$, nie są zatem dopisywane spacje w przypadku, gdy długość ciągu < ncols; jeżeli chcemy wypisać cały ciąg niezależnie od jego długości, to powinniśmy stosować stałą **GIANT**, np.

```
PWrite(10,10,GIANT,"ciąg testowy",A_NORM).
```

Alternatywny sposób wypisywania ciągów dostarcza funkcja **PSwrtcells**.

```
void PSwrtcells(row, col, captr, ncols)
/* wypisanie ciągu struktur CELL */
int row; /* zmieniany wiersz ekranu */
int col; /* pierwsza zmieniana kolumna */
CELL *captr; /* wskaźnik na ciąg struktur do zapisu na ekran */
int ncols; /* maksymalna liczba kolumn do zapisania */
```

Przydatność tej funkcji może budzić wątpliwość, stosowanie jej jest o wiele trudniejsze niż **PSwrite** (programista musi sam ustalić ciąg struktur do zapisu), a w zamian nie daje ona wielu dodatkowych możliwości. W praktyce funkcja ta jest wykorzystywana głównie nie przez programistów, ale przez wyższe warstwy oprogramowania. Stosowanie jej jawnie przez programistów jest ograniczone w zasadzie do przypadku, gdy chcą oni wypisać ciąg znaków o zmieniających się atrybutach.

Kolejna funkcja zmieniająca zawartość ekranu jest **PSfill**. Służy ona do wypełniania podanego prostokąta określonym znakiem:

```
PSfill(srect, chr, att) /* wypełnianie znakami prostokąta */
RECT *srect;           /* wskaźnik na wypełniany prostokąt */
char chr;              /* znak */
int att;               /* atrybut znaku */
```

Prostokąt ten jest przycinany do rozmiarów ekranu fizycznego. **PSfill** pozwala na wykonanie wielu często spotykanych działań, np.:

- czyszczenie ekranu: **PSfill(&GIANT_RECT, ' ', A_NORM)**

- zapelnienie wiersza o numerze 10 znakiem '*'

```
RECT r;
RASG(&r, 10, 0, 10, GIANT);
PSfill(&r, '*', A_NORM);
```

- narysowanie pionowej linii w 10 kolumnie, dzielącej cały ekran

```
RECT r;
RASG(&r, 0, 10, GIANT, 10);
PSfill(&r, C_V, A_NORM);
```

Ostatnią z funkcji zmieniających zawartość ekranu jest **PSslide**. Służy ona do przesuwania wybranego prostokąta.

```
BOOLEAN PSlide(srect, dir, dist)
/* przemieszczanie prostokąta */
RECT *srect /* przesuwany prostokąt */
int dir     /* dozwolone są 4 podstawowe kierunki:
              DIR_UP, DIR_DOWN, DIR_LEFT, DIR_RIGHT
              zdefiniowane w headerze display.h */
int dist    /* odległość, na jaką odbywa się przesunięcie */
```

Zarówno wejściowy, jak i wyjściowy prostokąt, są przycinane do rozmiarów ekranu fizycznego. Funkcja ta zwraca wartość **TRUE** lub **FALSE** w zależności od tego, czy dana implementacja zezwala na przemieszczanie prostokąta w wybranym kierunku, np. implementacja IBM/BIOS ignoruje polecenia przemieszczania w prawo lub lewo. Jeżeli

sytuacja taka występuje, to należy zastąpić jedno wywołanie *PSslide* sekwencją *PSwrite*. Może to spowolnić pracę oprogramowania, ale nie ogranicza znacząco możliwości programisty – analogiczna funkcja z poziomu ekranu wirtualnego *VSlide* jest wykonywana zawsze, niezależnie od ograniczeń implementacyjnych *PSslide*. Zawartość zwalnianego obszaru nie jest określona. Jeśli konieczne jest jego wypełnienie spacjami, to należy wywołać *PSfill*.

PSslide wykorzystuje się do wykonania często stosowanych operacji:

- skasowanie znaku w 10 wierszu i 5 kolumnie

```
RECT r;
RASG(&r,10,6,10,GIANT);
/* znak nr 6 jest pierwszym znakiem, który należy przesunąć */
PSslide(&r,DIR_LEFT,1);
PSwrite(10,PSwidth()-1,1," ",A_NORM);
/* konkatenacja spacji i ostatniego znaku */
- wstawienie wiersza przed wiersz o nr 10
```

```
RECT r;
RASG(&r,10,0,GIANT,GIANT);
PSslide(&r,DIR_DOWN,1);
RASG(&r,10,0,10,GIANT);
PSfill(&r,' ',A_NORM);
```

Kolejne funkcje spełniają już znacznie prostsze zadania: ustawianie położenia kursora (*PSsetcur*), wyświetlanie i usuwanie kursora (*PSshowcur*) oraz wydanie sygnału dźwiękowego (*PSbeep*).

```
void PSsetcur (row, col) /* ustawienie kursora */
int row; /* wiersz, w którym ma być ustawiony kursor */
int col; /* jak wyżej dla kolumny */
void PSshowcur (on)
/* wyświetlanie/kasowanie kursora */
BOOLEAN on; /* TRUE lub FALSE odpowiednio dla wyświetlenia
i zgaszenia kursora */
```

Uwaga. Zgaszony kursor może być przemieszczany przez funkcję *PSsetcur*.

```
void PSbeep() /* uaktywnienie sygnału dźwiękowego */
```

4.3.2. Analiza kodu funkcji

Zamieszczona w tym rozdziale analiza funkcji odnosi się do implementacji *IBM_mem*. Funkcje *PSwrite*, *PSfill* i *PSslide* zmieniają

zawartość bufora *dspbuf*. *PSsynch* zaś przenosi ten jego fragment do pamięci karty monitora, który jest konieczny do aktualizacji wyglądu ekranu. Fragment ten jest wyznaczony wskaźnikami *minptr* i *maxptr*:

```
static char *minptr; /* początek zmienianego obszaru */
static char *maxptr; /* koniec zmienianego obszaru */
```

Dodatkowo wprowadza się jeszcze zmienną *changed* sygnalizującą, czy ekran ma aktualną postać:

```
static BOOLEAN changed; /* TRUE, jeśli dokonano zmian, FALSE w pp.*/
```

Wszystkie te zmienne są lokalne i są niedostępne dla innych modułów. Poniżej przeanalizujemy sposób realizacji dwu funkcji: *PSwrite* i *PSfill*.

```
void PWrite(int row, int col, int ncols, char *str, int att)
```

```
{
    int i;
    short *p1=NULL, *p2;
    if (row<0 || row>MAXROW || col>MAXCOL)
        return;
    if (col+ncols>MAXCOL+1)
        ncols=MAXCOL+1-col;
    for (i=0; i<ncols && str[i]!='\0'; i++)
        if (col+i >=0)
        {
            p2=&dspbuf[row][col+i].word;
            if (p1==NULL)
                p1=p2;
            *p2=(att<<8) | (str[i]&0377);
        }
    if (p1!=NULL)
    {
        changed=TRUE;
        minptr=MIN(minptr,(char *) p1);
        maxptr=MAX(maxptr,(char *) p2);
    }
}
```

Po sprawdzeniu poprawności parametrów funkcja przepisuje kolejne znaki z atrybutem *att* do *dspbuf*. Aby przyspieszyć działanie, przepisuje się od razu oba bajty. Wskaźnik *p1* służy jedynie do sygnalizacji, czy został przepisany choć jeden znak i jednocześnie jest wskaźnikiem na pierwszy zmieniany bajt w *dspbuf*. Stosowanie operatora *&* w wyrażeniu *str[i] & 0377* ma na celu uniknięcie

propagowania ósmego bitu dla znaków o kodach przekraczających 127 (np. dla znaków tworzących ramki). Wskaźniki *minptr* i *maxptr* są zmieniane tylko wtedy, gdy zachodzi taka potrzeba.

```
void PSfill (RECT *srectp, char chr, int att)
{
    RECT srect;
    short word;
    int r;
    char *p1, *p2;
    if (!intersect(srectp, MAXROW, MAXCOL, &srect))
        return;
    changed=TRUE;
    word=(att<<8) | (chr & 0377);
    if (RWIDTH(&srect)==MAXCOL+1)
        repmem((char *)&dspbuf[srect.r1][0], (char *)&word, sizeof(CELL),
            RHEIGHT(&srect) * (MAXCOL+1));
    else
        if (RWIDTH(&srect)==1)
            for (r=srect.r1; r<=srect.r2; r++)
                dspbuf[r][srect.c1].word=word;
        else
            for (r=srect.r1; r<=srect.r2; r++)
                repmem((char *)&dspbuf[r][srect.c1], (char *)&word,
                    sizeof(CELL), RWIDTH(&srect));
    p1=(char *)&dspbuf[srect.r1][srect.c1];
    p2=(char *)&dspbuf[srect.r2][srect.c2];
    minptr=MIN(minptr, p1);
    maxptr=MAX(maxptr, p2);
}
```

Podczas analizy treści funkcji *PSfill* należy zwrócić uwagę na fakt, że w celu przyspieszenia jej działania oddzielnie są rozpatrywane trzy przypadki:

- prostokąt ma szerokość całego ekranu: wypełnianie za pomocą funkcji *repmem*
- prostokąt ma szerokość pojedynczego znaku: wypełnianie za pomocą operacji przypisania
- prostokąt nie mieści się w żadnej z powyższych grup: wypełnianie za pomocą kilku wywołań *repmem*.

4.3.3. Przykład

Poniższy program wyświetla na ekranie "odbijający się kwadrat". Do określenia kierunku przesuwu kwadratu służy generator liczb pseudolosowych. Kierunek przemieszczania jest utrzymywany aż do napotkania krawędzi ekranu. Liczba zmian kierunku jest wyświetlana w lewym górnym rogu ekranu.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <display.h>
#include <dsputil.h>
#include <pscreen.h>

#define GORA 0          /* dopuszczalne kierunki przesuwania */
#define DOL 1
#define PRAWO 2
#define LEWO 3
#define GORA_PRAWO 4
#define GORA_LEWO 5
#define DOL_PRAWO 6
#define DOL_LEWO 7

BOOLEAN ruch_do_dolu(RECT *okno)
{
    RECT r;
    if (okno->r2) == ROWS)
        return(FALSE);
    PSslide(okno, DIR_DOWN, 1);
    RASG(&r, okno->r1, okno->c1, okno->r1, okno->c2);
    PSfill(&r, C_HATCH, A_NORM);
    okno->r1++;
    okno->r2++;
    return(TRUE);
}

BOOLEAN ruch_w_prawo(RECT *okno)
{
    RECT r;
    if (okno->c2) == COLS)
        return(FALSE);
    PSslide(okno, DIR_RIGHT, 1);
    RASG(&r, okno->r1, okno->c1, okno->r2, okno->c1);
```



```

PSfill(&r,C_HATCH,A_NORM);
okno->c1++;
okno->c2++;
return(TRUE);
}

void pokaz_numer_zmiany(int nr)
{
char linia[20];
RECT okno_male;
int sz_okna=20;
RASG(&okno_male,0,COLS-sz_okna,0,COLS);
itoa(nr,linia,10);
PSfill(&okno_male,' ',A_INTENSE);
PWrite(0,COLS-sz_okna+(sz_okna-strlen(linia))/2,
      GIANT,linia,A_INTENSE);
}

void start (RECT *okno)
{
RECT o;
RASG(okno,1,0,ROWS/4,COLS/4);
PSbegin();
RASG(&o,0,0,GIANT,GIANT);
PSfill(&o,C_HATCH,A_NORM);
PSfill(okno,'+',A_INTENSE);
pokaz_numer_zmiany(0);
srand(17);
PSsynch();
}

void main()
{
int i;
int ruch_max=999;
int ruch_akt;
RECT okno;
start(&okno);
for (ruch_akt=0; ruch_akt<ruch_max; ruch_akt++)
{
switch (random(8))
{
case DOL:
while(ruch_do_dolu(&okno)!=TRUE)

```

```
        PSSynch();
        break;
    case PRAWO:
        while(ruch_w_prawo(&okno)==TRUE);
        PSSynch();
        break;
    case DOL_PRAWO:
        while(ruch_do_dolu(&okno) & ruch_w_prawo(&okno)==TRUE)
            PSSynch();
        break;
    ...
}
pokaz_numer_zmiany(++ruch_akt);
PSSynch();
}
PSend();
PSSynch();
}
```

Pominięte fragmenty kodu nie wnoszą nic istotnego, jedynie podczas przesuwania do góry program nie dopuszcza do zapisania górnego wiersza, w którym jest wyświetlany licznik. Jak widać, przesunięcia w kierunkach różnych od 4 podstawowych uzyskano składając przesunięcia w dostępnych kierunkach. Program ten jest umieszczony w pliku `ps_demo.c`.

4.4. Okna

Wiele pakietów oprogramowania zawiera funkcje do obsługi okien. Różnią się one bardzo możliwościami i dlatego konieczne jest na początku dokładne sprecyzowanie własności okien obsługiwanych przez funkcje biblioteki CTLS:

- okno nie może przekraczać rozmiarami ekranu fizycznego, jeżeli jest od niego mniejsze, to może (ale nie musi) być obwiedzione ramką
- wpisywanie danych do okna kończy się po dotarciu do jego prawego boku
- część lub całość okna może być zasłonięta innymi oknami, a podczas zmiany ich położenia odsłaniane fragmenty okna są odtwarzane automatycznie przez funkcję zarządzającą oknami na podstawie danych dostarczonych przez aplikację
- dokładnie jedno okno znajduje się na wierzchu, jest ono zawsze widoczne w całości

- z każdym oknem związany jest kursor, pokazywany jest jednak (jeżeli nie został on schowany) tylko kursor wierzchniego okna
- możliwe jest wpisywanie danych do dowolnego okna, niezależnie od jego wielkości, położenia na ekranie, nawet jeśli jest ono całkowicie zasłonięte przez inne okna
- funkcja zarządzająca oknami musi mieć możliwość ustalenia zawartości odsłanianego fragmentu okna
- maksymalna liczba okien nie może przekroczyć wartości stałej **MAXWIND**.

Funkcje tej warstwy mają możliwości większe niż w innych pakietach. Jedyne istotne ograniczenia to konieczność dostarczenia funkcji odtwarzającej zawartość okna oraz docinanie rozmiarów okien do rozmiarów ekranu fizycznego. Ograniczenia te są usunięte w warstwie ekranów wirtualnych.

4.4.1. Omówienie funkcji

Deklaracje poniższych funkcji są zamieszczone w headerze **window.h**. Obsługę ekranu na poziomie okien rozpoczyna wywołanie funkcji **Wbegin**

```
void Wbegin() /* inicjowanie pracy warstwy okien */
```

zaś kończy wywołanie funkcji **Wend**

```
void Wend() /* zakończenie pracy z warstwą okien */
```

Nie jest konieczne oddzielne stosowanie **PSbegin** i **PSend**. Okna tworzy się za pomocą funkcji **Wnew**:

```
int Wnew(redraw) /* utworzenie nowego okna */
```

```
int (*redraw)();
```

Funkcja ta alokuje pamięć na strukturę przechowującą informacje o oknie i zapisuje w niej wskaźnik na funkcję odtwarzającą zawartość okna. Jeżeli nawet ta niewielka ilość pamięci jest niedostępna, to zwracane jest 0 - numer okna o rozmiarze ekranu fizycznego, na jakie nakładane są wszystkie tworzone okna. Należy jednak podkreślić, że funkcja ta nie określa żadnych parametrów okna. Na to, aby możliwe było korzystanie z niego, konieczne jest wcześniejsze wywołanie funkcji **Wsetphys**. Numery kolejno tworzonych okien są zapisywane na stosie. Okno na jego szczycie jest zawsze pokazywane w całości. Kolejność umieszczania okien na stosie można zmienić funkcją **Wtop**.

Sposób odtwarzania jest uzależniony od sposobu wykorzystania okna i powinien być dostarczony przez aplikację. Funkcja **redraw** musi być zadeklarowana następująco:

```
int redraw(wnum, wrectp);
        /* funkcja odtwarzająca fragment okna */
int wnum;      /* numer okna */
RECT *wrectp /* odtwarzany fragment okna */
```

Pierwszy parametr pozwoli nam na stosowanie tej samej funkcji **redraw** do różnych okien. Prostokąt określany przez **wrectp** jest definiowany względem okna, a nie całości ekranu i określa najmniejszy fragment okna do odtworzenia. Czasami wygodne jest odtworzenie okna w całości. Funkcja ta jest wywoływana przez funkcje zarządzające oknami, wtedy gdy zachodzi potrzeba odtwarzania ich fragmentu na skutek przemieszczania, zmiany rozmiarów lub też kolejności umieszczenia okien na ekranie.

Jeżeli okna na siebie nie zachodzą, lub też odtwarzanie ich zawartości nie jest konieczne, to można parametrowi **redraw** nadać wartość **NULL**. Funkcja zarządzająca oknami nie odtwarza zawartości takich okien.

Okno może wielokrotnie zmieniać swoje rozmiary i położenie na ekranie. Służy do tego funkcja **Wsetphys**:

```
void Wsetphys(wnum, srectp)
        /* zmiana rozmiarów okna */
int wnum;      /* numer okna */
RECT *srectp;  /* współrzędne okna */
```

Współrzędne podaje się względem ekranu fizycznego i obejmują one ramkę, jeżeli okno ją ma. Do określania rodzaju, ramki jaka jest obwiedzione okno, służy funkcja **Wsetfrm**:

```
void Wsetfrm(wnum, frmtype, frmatt, title)
        /* określenie typu ramki */
int wnum;      /* numer okna */
FRMTYPE frmtype; /* rodzaj ramki */
int frmatt;     /* atrybut ramki */
char *naglowek; /* nagłówek ramki */
```

Istnieje 5 rodzajów ramek:

- **F_UNFRAMED**: okno nie ma w ogóle ramki; cały obszar wyznaczony przez drugi parametr **Wsetphys** jest udostępniony do zapisu
- **F_SINGLE**: w implementacjach na IBM PC/AT jest to ramka utworzona z pojedynczej linii
- **F_DOUBLE**: w implementacjach na IBM PC/AT jest to ramka utworzona z podwójnej linii

- **F_BLANK**: ramka ze spacji, przydatna na terminalach dysponujących znakami do rysowania linii ramki; jest ona szczególnie dobrze widoczna, gdy spacje są zapisywane z atrybutem **A_INVERSE**
- **F_AUTO**: ramka pokazywana jako **F_DOUBLE**, jeśli okno jest na wierzchu i jako **F_SINGLE** w przeciwnym przypadku; zmiana typu ramki następuje automatycznie.

Domyślnym typem ramki jest **F_SINGLE**. Typ **F_UNFRAMED** można stosować, gdy chcemy mieć okna o niestandardowej ramce. Tworzy się wówczas dwa okna bez ramek: większe - służące tylko do utworzenia ramki i nałożone na nie mniejsze - do wypisywania danych. Okna wymagają znikomej wielkości pamięci, jest to więc rozwiązanie w miarę efektywne i znacznie lepsze niż wypisywanie ramki za pomocą funkcji **PSwrite**.

Stale te są zdefiniowane w pliku nagłówkowym **window.h**. Jeżeli nie chcemy, by ramka miała tytuł, to parametr **title** powinien mieć wartość **NULL**, lub też musi on wskazywać na pusty ciąg.

Kasowanie okien uzyskujemy za pomocą funkcji **Wdispose**:

```
void Wdispose(wnum) /* kasowanie okna */
int wnum;          /* numer usuwanego okna */
```

Funkcja **Wdispose** powoduje usunięcie okna z ekranu i zwolnienie pamięci zajmowanej dotychczas przez jego opis. Jeżeli funkcja **redraw** została określona, to okna które ono poprzednio zasłaniała, zostają odtworzone.

Funkcje zmieniające zawartość okna są bardzo podobne do funkcji zmieniających zawartość ekranu fizycznego. Jedyna istotna różnica polega na tym, że wszystkie wielkości podaje się względem okna a nie ekranu:

```
void Wwrite(wnum, wrow, wcol, ncols, str, att)
/* wypisywanie ciągu znaków */
int wnum; /* numer okna */
int wrow; /* zmieniany wiersz okna */
int wcol; /* pierwsza zmieniana kolumna */
int ncols; /* maksymalna liczba znaków do wpisania */
char *str; /* zapisywany ciąg */
int att; /* atrybut zapisywanych znaków */

void Wwrtcells(wnum, row, col, captr, ncols)
/* wypisywanie ciągu struktur CELL */
int wnum; /* numer okna */
int row; /* zmieniany wiersz okna */
```

```

int col;          /* pierwsza zmieniana kolumna */
CELL *captr;     /* wskaźnik na ciąg struktur do zapisu na ekran */
int ncols;       /* maksymalna liczba znaków do wpisania */

void Wfill(wnum, wrectp, chr, att)
    /* wypełnienie prostokąta wewnątrz okna */
int wnum;        /* numer okna */
RECT *wrectp;    /* wskaźnik na wypełniany prostokąt */
char chr;        /* znak */
int att;         /* atrybut znaku */

BOOLEAN Wslide(wnum, wrectp, dir, dist)
    /* przemieszczanie prostokąta wewnątrz okna */
int wnum;        /* numer okna */
RECT *wrectp     /* przesuwany prostokąt */
int dir          /* dozwolone są 4 podstawowe kierunki:
                  DIR_UP, DIR_DOWN, DIR_LEFT, DIR_RIGHT
                  zdefiniowane w headerze display.h */
int dist         /* odległość, na jaką odbywa się przesunięcie */

```

W odróżnieniu od wszystkich pozostałych funkcja *Wslide* wymaga, aby okno, na którym jest wykonywana operacja, było na szczycie. Efekty działania wszystkich tych funkcji są ograniczane do jednego okna.

Do ustawiania oraz wyświetlania/gaszenia kursora służą funkcje *Wsetcur* i *Wshowcur*:

```

void Wsetcur (int wnum, row, col)
    /* ustawianie kursora w oknie */
int wnum;     /* numer okna */
int row;      /* wiersz, w którym ma być ustawiony kursor */
int col;      /* jak wyżej dla kolumny */

void Wshowcur (int wnum, on) /* kasowanie/wyświetlanie kursora */
int wnum;     /* numer okna */
BOOLEAN on;   /* TRUE lub FALSE odpowiednio dla wyświetlenia
               i zgaszenia kursora */

```

Każde okno ma swój własny kursor, wyświetlany jest on jednak tylko dla okna będącego na szczycie. Jeśli nakażemy jego usunięcie, to kursor w ogóle nie będzie widoczny na ekranie.

Funkcja *Wtop* pozwala na zmianę kolejności zapisu okien na stosie:

```

void Wtop(wnum) /* przesunięcie okna na szczyt */
int wnum;      /* numer okna */

```

Funkcja ta wymusza umieszczenie na szczycie stosu okna o numerze `wnum`. Wykonanie jej bywa czasochłonne, bowiem może wiązać się z koniecznością odtworzenia zawartości wielu okien funkcją `redraw`.

Funkcja `Wgettop` pozwala na określenie numeru okna umieszczonego na szczycie stosu:

```
int Wgettop()      /* podaje numer wierzchniego okna */
```

zaś funkcja `Wshuffle` na przemieszczenie okna ze szczytu stosu na ostatnią pozycję:

```
void Wshuffle()    /* przesunięcie okna ze szczytu na dół stosu */
```

Programista ma ponadto możliwość usunięcia okna z ekranu, bez jego kasowania:

```
void Whide(wnum)   /* schowanie okna */
```

```
int wnum;          /* numer okna */
```

Schowane w ten sposób okno jest dostępne dla wszystkich pozostałych funkcji, możliwe jest nawet wpisanie do niego danych. Pojawienie się okna ponownie na ekranie można uzyskać za pomocą funkcji `Wtop` lub `Wsetphys`.

Ostatnia grupa funkcji pozwala na powiększenie okna tak, by jego treść (bez ramki) zajmowała cały ekran (`Wzoom`), przywrócenie go do poprzednich rozmiarów (`Wunzoom`) oraz sprawdzenie, czy okno zostało w ten sposób powiększone (`Wiszoomed`):

```
void Wzoom(int wnum)
```

```
    /* powiększenie okna do rozmiarów całego ekranu */
```

```
void Wunzoom(int wnum)
```

```
    /* przywrócenie "normalnych" rozmiarów okna */
```

```
BOOLEAN Wiszoomed(int wnum)
```

```
    /* czy okno jest powiększone? */
```

Jak widać, funkcje realizujące warstwę okien są dość wszechstronne i pozwalają na wykonanie wszystkich podstawowych operacji na oknach. Podczas programowania na tym poziomie zmiana zawartości ekranu funkcjami z poziomu ekranu fizycznego powoduje najczęściej niepożądane efekty uboczne. Spowodowane jest to tym, że funkcja do odtwarzania zawartości okna (`redraw`) najczęściej nie jest w stanie uwzględnić zmian wprowadzonych przez funkcje z niższego poziomu. Jedynymi funkcjami z warstwy ekranu fizycznego, które należy stosować, jest naturalnie funkcja do uaktualniania wyglądu ekranu (`FSsynch`) oraz funkcje do określania rozmiarów kursora (`BVsetctype`).

Znacznym utrudnieniem, napotykanym podczas posługiwania się warstwą okien, jest konieczność podania funkcji **redraw** przy tworzeniu okna. Należy jednak podkreślić, że poprawne zdefiniowanie tej jednej funkcji pozwala na wykonanie wszystkich operacji na oknach. Wykonanie jednej funkcji warstwy okien, np. **Wtop**, powoduje automatyczne wywołanie **redraw** dla tych okien, których wygląd na ekranie ulegnie zmianie w wyniku wykonania operacji. Programy aplikacyjne, po dostarczeniu funkcji **redraw** w momencie tworzenia okna, nie muszą już interesować się sposobem nakładania się okien na ekranie.

Podczas implementacji funkcji **redraw** należy pamiętać, że musi ona zawsze dysponować aktualnymi danymi o zawartości okna. Oznacza to, że funkcje zmieniające zawartość lub też rozmiar okna muszą w większości przypadków dokonywać równocześnie zmian w buforze zawierającym treść okna (patrz funkcja **VSwrite** z warstwy ekranów wirtualnych). Celowe jest opracowanie kilku takich funkcji, wykorzystywanych do definiowania różnych rodzajów okien. Przydatne jest zwłaszcza opracowanie funkcji dla okien wyświetlających znaki, za pomocą tylko jednego atrybutu. Pozwala to na osiągnięcie komfortu pracy podobnego do tego, jaki jest właściwy dla warstwy ekranów wirtualnych, przy znacznie mniejszej wielkości wykorzystywanej pamięci operacyjnej.

Prosty, a jednocześnie uniwersalny sposób implementacji tej funkcji, polega na przypisaniu każdemu oknu bufora z jego treścią. Sposób ten nie jest tu bliżej omawiany, jest on bowiem wykorzystany w warstwie ekranów wirtualnych. Jeżeli założymy, że wszystkie znaki są wypisywane z tym samym atrybutem, to rozmiar tego bufora może być o połowę mniejszy niż w przypadku ekranów wirtualnych. W sytuacji, gdy okna się na siebie nie nakładają, bądź też są statyczne (tj. nie zmieniają swej kolejności i rozmiarów), podawanie funkcji **redraw** jest zbędne.

4.4.2. Analiza kodu funkcji

Struktura **WINDOW**, zdefiniowana w pliku nagłówkowym **window.h**, zawiera wszystkie dane o oknie, pozwalające na wykonanie wymienionych, skomplikowanych działań.

```
typedef struct
```

```
{  
    RECT srect;           /* położenie okna na ekranie fizycznym */  
    RECT svsrc;           /* kopia srect (dla powiększania) */  
    FRMTYPE frmtype;      /* typ ramki */  
    short frmatt;         /* atrybut ramki */  
};
```



```

char *title;          /* tytuł ramki */
FRMTYPE svfrmttype;   /* kopia ramki (dla powiększania) */
BOOLEAN zoomed;       /* czy okno jest powiększone? */
short currow;         /* położenie kursora: wiersz */
short curcol;         /* położenie kursora: kolumna */
BOOLEAN showcur;      /* czy kursor ma być pokazany? */
int (*drawfcn)();     /* funkcja odtwarzania zawartości */
) WINDOW;

```

Plik ten zawiera też **MAXWIND** – stałą określającą maksymalną liczbę okien:

```
#define MAXWIND 7
```

W pliku `window.c` zdefiniowana jest statyczna tablica, opisująca wszystkie możliwe do powołania okna:

```
static WINDOW *wind[MAXWIND];
```

oraz zmienne **maxrow** i **maxcol**, określające maksymalną liczbę wierszy i kolumn okna. Tablica **wind** jest użyta do implementacji stosu wskaźników na opisy okien. Porządek elementów stosu wyznacza kolejność nakładania okien na ekran.

Warstwa obsługi okien jest dość rozbudowana, kod źródłowy jej funkcji zajmuje ponad 15 KB. Omówienie go w całości wykracza poza ramy niniejszej pracy. Z tego powodu zdecydowaliśmy się na prezentację najciekawszego – naszym zdaniem – fragmentu kodu, odpowiedzialnego za ustalenie, które fragmenty okien mają być widoczne na ekranie.

Programista może wpisywać dane do okna funkcją **Wwrite** niezależnie od tego, czy jest ono na szczycie, czy też nie. Oznacza to, że funkcje implementujące warstwę okien muszą określić, jaki fragment wypisywanego ciągu jest widoczny na ekranie. Może się tak bowiem zdarzyć, że jego część lub nawet całość, będzie zasłonięta przez inne okna. Podobnie po zmianie położenia lub wielkości okna konieczne jest określenie fragmentów okien, jakie należy odtworzyć funkcją **redraw**. Wszystkie te operacje są wykonywane bardzo często, konieczne jest zatem znalezienie rozwiązania pozwalającego na ich szybką realizację.

Przyjęte rozwiązanie spełnia te wymagania. Polega ono na wykorzystaniu dwuwymiarowej tablicy znaków o nazwie **mask**, mającej wymiary ekranu fizycznego. Wartość znaku wskazuje numer okna, do którego należy odpowiadająca mu pozycja na ekranie. Najczęściej stosowane funkcje (**Wwrite**, **Wslide**, **Wfill**) powinny być wykonywane jak najszybciej. Funkcje te korzystają z tablicy **mask** lecz nie zmieniają jej zawartości. Modyfikacji tej tablicy dokonuje się jedynie dla funkcji zmieniających rozmiary lub położenie okien, np. **Wzoom**.

Tablica *mask* jest tworzona przez funkcję *allocmask*:

```
static char **mask=NULL;
static void allocmask()
{
    int row;
    mask=(char **)xmalloc((maxrow+1)*sizeof(char *));
    for (row=0; row<=maxrow; row++)
        mask[row]=xmalloc(maxcol+1);
}
```

Jak widać, zmienna *mask* tworzy tablicę złożoną z tylu wskaźników, ile jest wierszy w ekranie fizycznym. Każdy z nich wskazuje na ciąg znaków o długości równej liczbie kolumn. Takie dynamiczne określenie rozmiarów tablicy pozwala na stosowanie tego samego kodu wynikowego do obsługi ekranów fizycznych o różnych rozmiarach. Funkcja *allocmask* jest wywoływana przez funkcję *Wbegin*. Funkcja *freemask* zwraca całą pamięć przeznaczoną na zapisanie tablicy *mask*.

```
static void freemask()
{
    int row;
    if (mask!=NULL)
        for(row=0; row<=maxrow; row++)
            free(mask[row]);
    free((char*)mask);
    mask=NULL;
}
```

Jest ona wywoływana przez funkcję *Wend*. Kolejna funkcja, *setmask*, wypełnia żądany prostokąt numerem okna:

```
static void setmask(wnum,srectp)
int wnum; /* numer okna */
RECT *srectp; /* prostokąt do wypełnienia */
{
    RECT srect;
    int r;
    if (!intersect(srectp,maxrow,maxcol,&srect))
        return;
    for (r=srect.r1; r<=srect.r2; r++)
        if (srect.c2>=srect.c1)
            setmem(&mask[r][srect.c1],RWIDTH(&srect),wnum);
}
```

Parametr *srectp* podaje się względem ekranu fizycznego. Prostokąt **srectp* jest docinany do wymiarów ekranu, a następnie zostaje on wypełniony funkcją *setmem*.

Po wykonaniu jakiegokolwiek funkcji zmieniającej rozmiary lub położenie okien, wywoływana jest funkcja *reset*, ustalająca aktualną postać wybranego fragmentu tablicy *mask*.

```
static void reset(RECT *srectp)
{
    RECT srect, srect2;
    WINDOW *w;
    int i;
    if (srectp->r1>srectp->r2 || srectp->c1>srectp->c2)
        return;
    if (!intersect(srectp,maxrow,maxcol,&srect))
        return;
    for (i=0; i<numwinds; i++)
    {
        w=wind[order[i]];
        RASG(&srect2,MAX(srect.r1,w->srect.r1),
            MAX(srect.c1,w->srect.c1), MIN(srect.r2,w->srect.r2),
            MIN(srect.c2,w->srect.c2));
        setmask(order[i],&srect2);
    }
    draw(&srect);
}
```

Na początku funkcja ta sprawdza, czy przekazany - jako parametr prostokąt **srectp* - zawiera choć jeden znak z ekranu fizycznego oraz docina go do jego rozmiarów. Zabezpieczenie zapewni, że wpisywane dane nie przekroczą rozmiarów zdefiniowanych tablic nawet wtedy, gdy błędnie działają inne fragmenty kodu. Funkcja ta nadzoruje zapisywanie dużych fragmentów pamięci i, gdy są błędnie podane parametry, jest to operacja potencjalnie niebezpieczna. Następnie wykonywana jest pętla, w której przeglądane są okna, począwszy od okna znajdującego się na końcu stosu. Dla każdego z nich ustalana jest część wspólna przypisanego mu okna i badanego fragmentu ekranu, po czym wpisywany jest do niej numer odpowiedniego okna.

Zadaniem funkcji *draw* jest określenie jakie okna są widoczne w zadanym prostokącie, a następnie odtworzenie dla każdego z widocznych tam okien jego ramki i zawartości. Ramka jest wyświetlana na podstawie danych zapisanych w odpowiedniej strukturze *WINDOW*, zaś jego wnętrze - przez wywołanie funkcji odtwarzającej zawartość okna.

4.4.3. Przykład

Poniższy program tworzy 6 nie zachodzących na siebie okien i wpisuje do nich kolejne liczby. W każdym wierszu jest zapisana jedna liczba. Liczby te są wyświetlane z atrybutami `A_NORM` i `A_INTENSE`.

```
#include <stdio.h>
#include <stdlib.h>
#include <display.h>
#include <dsputil.h>
#include <pscreen.h>
#include <window.h>
#include <port.h>

static void wd_demo_start(void);

void WdPodajKursor(int wnum, int *wrow, int *wcol)
{
    WINDOW *w=wind[wnum];
    *wrow=w->currow;
    *wcol=w->curcol;
}

void Wnewline(int wnum, char *txt, int att)
{
    int row, col, top_window;
    RECT okno;
    Wgetphys(wnum, &okno, FALSE);
    WdPodajKursor(wnum, &row, &col);
    if ((row==(RHEIGHT(&okno)-1)) && (col==(RWIDTH(&okno)-1)))
    {
        top_window=Wgettop();
        Wtop(wnum);
        Wslide(wnum, &GIANT_RECT, DIR_UP, 1);
        Wtop(top_window);
    }
    Wwrite(wnum, row, 0, GIANT, txt, att);
    if (++row==(RHEIGHT(&okno)))
        Wsetcur(wnum, --row, RWIDTH(&okno)-1);
    else
        Wsetcur(wnum, row, 0);
}
```

```

static void wd_demo_start()
{
    RECT o;
    int nr;
    RASG(&o,0,0,6,50);
    nr=Wnew(NULL);
    Wsetphys(nr,&o);
    Wsetfrm(nr,F_AUTO,A_NORM,"okno nr 0");
    RASG(&o,0,51,8,78);
    ...
    RASG(&o,21,0,24,78);
    nr=Wnew(NULL);
    Wsetphys(nr,&o);
    Wsetfrm(nr,F_AUTO,A_NORM,"okno nr 6");
}

void main()
{
    int i;
    char txt[COLS+1];
    Wbegin();
    wd_demo_start();
    PSsynch();
    for (i=0; i<1000; ++i)
    {
        int ncols;
        RECT okno;
        Wgetphys(Wgettop(),&okno,FALSE);
        ncols=RWIDTH(&okno);
        sprintf(txt,"%*d*s",ncols>>1,i,ncols-(ncols>>1)," ");
        Wnewline(Wgettop(),txt, (i&1)? A_NORM:A_INVERSE);
        Wshuffle();
        PSsynch();
    }
    Wend();
    PSsynch();
}

```

Pominięty fragment kodu definiuje okna od 1 do 5 włącznie. Na uwagę zasługuje kilka fragmentów tego programu:

1. Biblioteka CTLS nie zawiera funkcji podającej położenie kursora w oknie. Jeżeli dysponujemy kodem źródłowym funkcji, to najwygodniej jest dopisać funkcję zwracającą odpowiedni fragment opisu okna. Jeśli tak nie jest, to musimy sami zadbać o zapamiętanie położenia kursora. W tym przykładzie zadanie to spełnia funkcja **Wdustawkursor**. Jest to konieczne, bowiem tablica **wind** jest lokalna dla warstwy **window.c**.
2. Funkcja **Wnewline** traktuje znak wskazywany przez kursor jako pierwszy znak, gdzie ma być dokonany zapis. Ponieważ kursora nie może ustawić poza obrębem okna, jeden ze znaków w jego wnętrzu musi sygnalizować jego zapełnienie. Wybrany do tego celu znak jest położony w prawym dolnym rogu okna. Oznacza to, że znaku tego nie można zapisać. Podobne rozwiązanie przyjęto także w innych pakietach, np. w Borland C.
3. Przed przystąpieniem do "podciągania", okna należy umieścić na szczycie.
4. Na to, aby funkcja **Wnewline** stała się użytecznym narzędziem programisty, należy ją rozbudować tak, by: pozwalała na wpisywanie tekstu nie tylko od początku wiersza, ale także od dowolnego miejsca, nie zmieniała kolejności ułożenia okien (jest to efekt uboczny przesuwania ekranu), obsługiwała wieloliniowe teksty i znaki tabulacji. Modyfikacje te pozostawiono do wykonania Czytelnikowi w charakterze ćwiczenia.

Program ten jest umieszczony w pliku **wd_demo.c**.

4.5. Ekran wirtualne

Kolejną warstwę oprogramowania ekranu tworzą ekrany wirtualne. Przed przystąpieniem do prezentacji funkcji realizujących tę warstwę omówimy pokrótce podstawowe ich własności:

- użytkownik może dysponować wieloma ekranami wirtualnymi
- wielkość ekranu wirtualnego nie jest ograniczona rozmiarami monitora, ale jedynie wielkością dostępnej pamięci operacyjnej
- niemożliwe jest zmienienie rozmiarów ekranu po jego utworzeniu
- z każdym ekranem związane jest jedno okno, służące do wyświetlania jego zawartości
- w przeciwieństwie do warstwy okien, funkcja **redraw** jest dostarczana automatycznie przez tę warstwę

- każdy ekran ma kursor, okno jest automatycznie przesuwane w ten sposób po zawartości ekranu, by był on zawsze widoczny.

Jak widać, możliwości ekranów wirtualnych są znacznie większe niż możliwości okien. Cenę, jaką płaci się za stosowanie ekranów wirtualnych, jest zwiększenie wielkości wymaganej pamięci do obsługi tej warstwy (ok. 8 KB na kod funkcji oraz pamięć na zapamiętanie zawartości ekranu) oraz nieznaczne spowolnienie szybkości działania programu.

4.5.1. Omówienie funkcji

Deklaracje poniższych funkcji są zamieszczone w headerze `vscreen.h`. Sposób korzystania z warstwy ekranów wirtualnych jest bardzo zbliżony do sposobu, jaki znamy z poprzednich warstw. Prace rozpoczyna wywołanie `VSbegin`:

```
void VSbegin(); /* początek pracy warstwy ekranów wirtualnych */
zaś kończy VSend:
```

```
void VSend(). /* koniec pracy warstwy ekranów wirtualnych */
```

Do tworzenia nowego ekranu stosuje się funkcję `VSnew`:

```
int VSnew(nrows, ncols, srectp, title, att)
/* tworzenie nowego ekranu wirtualnego */
int nrows;      /* liczba wierszy */
int ncols;     /* liczba kolumn */
RECT *srectp;  /* położenie i rozmiar okna */
char *title;   /* nagłówek okna */
int att;       /* atrybut wyświetlania nagłówka okna */
```

Funkcja ta zwraca numer ekranu. W przypadku wystąpienia błędu jest nim stała `BADVSCREEN` o wartości `-1`, zdefiniowana w pliku nagłówkowym `vscreen.h`.

Tworzony ekran nie musi mieć rozmiarów zgodnych z rozmiarami przypisanego mu okna. Jeżeli jest od niego mniejszy, to pokazywany jest w jego górnym lewym rogu. Jeżeli jest większy, to okno przedstawia jedynie jego fragment – początkowo jest to lewy górny róg. Podczas tworzenia ekranu alokowana jest pamięć na bufor jego treści. Wielkość tego bufora wynosi `ncols * nrows * sizeof(CELL)`.

Jeżeli parametr `title` ma wartość `NULL`, to okno nie będzie miało ramki, w przeciwnym przypadku będzie to ramka pojedyncza. Ramkę pojedynczą bez nagłówka uzyskuje się, gdy parametr `title` wskazuje na pusty ciąg. Nie istnieje żadna korelacja między numerem ekranu wirtualnego a numerem przypisanego mu okna.

Omówienie zasad określania wielkości ekranów wirtualnych znaleźć można w p. 4.6.

Ekran y wirtualne usuwa się funkcją *VSdispose*:

```
void VSdispose(vsnun) /* kasowanie ekranu wirtualnego */
int vsnun;           /* numer usuwanego ekranu */
```

Rozmiary ekranów wirtualnych nie ulegają zmianie. Jeżeli jednak piszemy uniwersalne funkcje, mające działać na dowolnym ekranie, to przydatne jest określanie wielkości ekranu tylko na podstawie jego numeru. Służy do tego funkcja *VSgetsize*:

```
void VSgetsize(vsnun, nrowsp, ncolsp)
/* określenie rozmiarów ekranu wirtualnego */
int vsnun; /* numer ekranu wirtualnego */
int *nrowsp; /* liczba wierszy ekranu */
int *ncolsp; /* liczba kolumn ekranu */
```

Następujące funkcje są odpowiednikami funkcji znanych z poprzednich warstw oprogramowania: *VSwrite*, *VSfill*, *VSslide*, *VSsetcur*, *VSshowcur*:

```
void VSwrite(vsnun, vrow, vcol, ncols, str, att)
/* wypisywanie ciągu znaków */
int vsnun; /* numer ekranu wirtualnego */
int vrow; /* numer wiersza */
int vcol; /* numer kolumny */
int ncols; /* długość ciągu znaków */
char *str; /* wypisywany ciąg znaków */
int att; /* atrybut wypisywanego ciągu */
```

Jedyna różnica między *VSwrite* a *PSwrite* i *Wwrite* polega na tym, że, gdy $ncols \neq \text{GIANT}$ i $ncols > \text{strlen}(str)$, wówczas do ciągu *str* dopisywanych jest $ncols - \text{strlen}(str)$ spacji. Oznacza to, że wywołanie funkcji z $ncols = \text{GIANT} + 1$ spowoduje, jeżeli jest to konieczne, dopełnienie wiersza spacjami.

```
void VSfill(vsnun, vrectp, char chr, int att)
/* wypełnianie prostokąta */
int vsnun; /* numer ekranu wirtualnego */
RECT *vrectp; /* prostokąt do wypełnienia */
char chr; /* znak wypełniający prostokąt */
int att; /* atrybut znaku chr */
```



```

void VSslide(vnum, vrectp, dir, dist)
    /* przemieszczanie prostokąta */
int vnum;    /* numer ekranu wirtualnego */
RECT *vrectp; /* prostokąt do przesunięcia */
int dir;     /* kierunek przesunięcia */
int dist;    /* odległość */

```

Należy zwrócić uwagę, że funkcja **VSslide** jest typu void. Funkcja ta zawsze przemieszcza prostokąt, niezależnie od tego czy jest to wykonywane w warstwie fizycznej przez **PSslide**, czy też dopiero w warstwie ekranów wirtualnych.

```

void VSsetcur(vnum, vrow, vcol)
    /* ustawianie położenia kursora */
int vnum;    /* numer ekranu wirtualnego */
int vrow;    /* wiersz */
int vcol;    /* kolumna */

```

VSsetcur jest znacznie bardziej skomplikowaną funkcją niż w poprzednich warstwach. Oprogramowanie tej warstwy zapewnia, że kursor zawsze znajduje się w obrębie okna ekranu wirtualnego. Występuje to nawet wtedy, gdy został on zgaszony. Jeżeli kursor zostanie ustawiony w takim miejscu ekranu, które nie mieści się aktualnie w ramach okna, to okno zostanie automatycznie przesunięte, tak by mógł być on widoczny. Okno można przesuwac także w sposób jawny, za pomocą funkcji **VSpan**.

```

void VSshowcur(vnum, on)
    /* wyświetlanie/gaszenie kursora */
int vnum;    /* numer ekranu wirtualnego */
BOOLEAN on;  /* zapal/zgaś kursor */

```

Kolejne funkcje są funkcjami specyficznymi dla tej warstwy. Pierwszą z nich jest funkcja do określania położenia kursora w ekranie wirtualnym **VSgetcur**:

```

void VSgetcur(vnum, vrowp, vcolp)
    /* ustalenie położenie kursora */
int vnum;    /* numer ekranu wirtualnego */
int *vrowp;  /* wiersz, w którym znajduje się kursor */
int *vcolp;  /* kolumna, w której znajduje się kursor */

```

Kolejna funkcja **VSgetwloc** podaje położenie lewego górnego rogu okna względem ekranu wirtualnego. Należy pamiętać, że jest to położenie okna względem ekranu wirtualnego, a nie fizycznego.

```
void VSgetwloc(vsnum, vrowlp, vcollp)
    /* ustalenie położenia okna w ramach ekranu wirtualnego */
*/
int vsnum;    /* numer ekranu wirtualnego */
int *vrowlp;  /* położenie lewego górnego rogu okna */
int *vcollp;  /* położenie lewego górnego rogu okna */
```

Jedną z najbardziej interesujących funkcji jest funkcja **VSpan**, służąca do przemieszczania okna po ekranie wirtualnym:

```
void VSpan(vnum, dir, dist)
    /* przesuwanie okna po ekranie wirtualnym */
int vsnum;    /* numer ekranu wirtualnego */
int dir;      /* kierunek przemieszczenia */
int dist;     /* odległość */
```

VSpan, w przeciwieństwie do **VSlide**, nie modyfikuje zawartości ekranu wirtualnego, a jedynie zmienia jego fragment oglądany w oknie. Jeżeli zachodzi taka potrzeba, to automatycznie przesuwany jest również kursor.

Pomimo tego, że do przeglądania ekranów wirtualnych wykorzystywane są okna, oprogramowanie tej warstwy nie zawiera funkcji do wykonywania wszystkich działań na oknach. Korzystanie z funkcji zmieniających bezpośrednio zawartość okna (**Wwrite**, **Wslide**) jest jednak niedopuszczalne. Wpisują one dane do okien poza kontrolą warstwy ekranów wirtualnych i tym samym powodują nieprawidłowe odtwarzanie zawartości okien. W wielu wypadkach przydatne jest natomiast stosowanie funkcji: **Wtop**, **Wshuffle**, **Wsetphys**, **Wsetfrm**, **Wzoom**, **Wunzoom**, **Whide**. Na to, aby móc z nich skorzystać, konieczne jest jednak określenie numeru okna związanego z zadany ekranem wirtualnym. Służy do tego funkcja **VSgetwnum**:

```
int VSgetwnum(vsnum) /* ustalenie nr okna przypisanego ekranowi */
int vsnum;           /* numer ekranu wirtualnego */
```

Znacznie rzadziej korzysta się z funkcji **VSgetvsnum**, zwracającej numer ekranu wirtualnego, oglądanego za pomocą określonego okna:

```
int VSgetvsnum(int wnum) /* ustalenie nr ekranu związanego z oknem */
int wnum;               /* numer okna */
```

Zwykle numery ekranów wirtualnych i przypisanych im okien nie są sobie równe. Pomylenie tych dwu numerów powoduje błędne działanie funkcji tej warstwy.

4.5.2. Analiza kodu funkcji

Struktura *VScreen*, zdefiniowana w pliku *vscreen.c*, zawiera wszystkie dane o ekranie wirtualnym, potrzebne do realizacji funkcji tej warstwy.

```
typedef struct
(
    short winnum;      /* numer okna przypisanego do ekranu */
    short vwinrow1;    /* położenie okna na ekranie wirtualnym, wiersz */
    short vwincol1;    /* położenie okna na ekranie wirtualnym, kolumna */
    short vrow2;       /* maksymalny numer wiersza ekranu wirtualnego */
    short vcol2;       /* maksymalny numer kolumny ekranu wirtualnego */
    short vcurrow;     /* położenie kursora w na ekranie wirtualnym,
                       wiersz */
    short vcurcol;     /* położenie kursora w na ekranie wirtualnym,
                       kolumna */
    CELL **ca;         /* wskaźnik na bufor z zawartością ekranu */
) VSCREEN;
```

Jak widać, umieszczono tu tylko niezbędne dane. Jeżeli zachodzi taka potrzeba, to informacje o wielkości i położeniu okna (jak i kursora) są pobierane z warstwy realizującej okna. Pozwala to na uniknięcie redundancji w zapisie danych, a ponadto jest przejawem niezależności ekranu wirtualnego od przypisanego mu okna.

Ostatnim elementem definicji tej struktury jest *ca* – wskaźnik na tablicę wskaźników do tablicy struktur *CELL*. Przyjęcie takiego rozwiązania pozwala na obejście ograniczenia, jakim jest niemożność określania rozmiarów tablic w czasie wykonywania programu.

Maksymalna liczba ekranów jest równa maksymalnej liczbie okien:

```
#define MAXVSCREEN    MAXWIND
```

Tablica *vscreen* zawiera wskaźniki na struktury opisujące utworzone ekrany wirtualne:

```
static VSCREEN *vscreen[MAXVSCREEN];
```

W rozdziale tym przeanalizujemy działanie dwóch funkcji: *VWrite*, wpisującej dane do bufora oraz *redraw*, służącej do odtwarzania okien na podstawie tego bufora, zgodnie z wymaganiami określonymi w p. 4.4. Uproszczona wersja tej funkcji może Czytelnikom posłużyć do napisania analogicznej funkcji dla warstwy okien:

```

void VSwrite(int vsnum, int vrow1, int vcol1, int ncols,
             char *str, int att)
{
    VSCREEN *vs;
    CELL *cellp;
    int i;
    BOOLEAN pad;
    if (vsnum==BADVSCREEN)
        return;
    vs=vscreen[vsnum];
    if (vrow1<0 || vrow1>vs->vrow2 || vcol1<0 || vcol1>vs->vcol2)
        return;
    pad=ncols!=GIANT;
    if(vcol1+ncols > vs->vcol2+1)
        ncols=vs->vcol2-vcol1+1;
    for (i=0; i<ncols && str[i]!='\0'; i++)
    {
        cellp=&vs->ca[vrow1][vcol1+i];
        cellp->chr=str[i];
        cellp->att=att;
    }
    if(pad && i<ncols)
        repcell(&vs->ca[vrow1][vcol1+i], ' ',att,ncols-i);
    else
        ncols=i;
    if(vcol1<vs->vwincol1)
    {
        ncols-=vs->vwincol1-vcol1;
        vcol1=vs->vwincol1;
    }
    Wwrtcells(vs->winnum,vrow1-vs->vwinrow1,vcol1-vs->vwincol1,
             vs->ca[vrow1][vcol1],ncols);
}

```

Jak widać, na początku funkcja sprawdza poprawność przekazanych parametrów, tj. kontroluje, czy ekran wirtualny o podanym numerze został już utworzony oraz, czy miejsce zapisu mieści się w jego ramach. Następnie ustalane jest, czy wymagane jest uzupełnienie ciągu spacjami oraz określa się maksymalną długość ciągu, jaki można zapisać w wierszu od podanego miejsca. Wpisywanie danych realizuje instrukcja `for`, a uzupełnianie ciągu spacjami funkcja `repcell`. Modyfikacji

zawartości okna dokonuje się jednym wywołaniem **Wwrtcells**. Przyspieszenie obsługi ekranów wirtualnych jest główną przyczyną implementacji dwu funkcji zapisu danych do okna: **Wwrite** i **Wwrtcells**.

```
static int redraw(int wnum, RECT *wrectp)
{
    VSCREEN *vs;
    RECT srect, wrect, wrect2;
    int vrow1, vcol1, vrow2, vcol2, vr, vc2;
    Wgetphys(wnum,&srect,FALSE);
    if (!intersect(wrectp,RHEIGHT(&srect)-1,RWIDTH(&srect)-1, &wrect))
        return(0);
    vs=vscreen[VSgetvsnum(wnum)];
    vrow1=vs->vwinrow1+wrect.r1;
    vcol1=vs->vwincol1+wrect.c1;
    vrow2=vs->vwinrow1+wrect.r2;
    vcol2=vs->vwincol1+wrect.c2;
    for (vr=vrow1; vr<=vrow2; vr++)
    {
        if (vr>vs->vrow2)
        {
            wrect.r1=vr- vs->vwinrow1;
            Wfill(wnum,&wrect,' ',A_NORM);
            break;
        }
        vc2=MIN(vcol2,vs->vcol2);
        Wwrtcells(wnum,vr-vs->vwinrow1, wrect.c1, &vs->ca[vr][vcol1],
            vc2-vcol1+1);
        vc2=MAX(vcol1,vc2+1);
        if (vc2<=vcol2)
        {
            RASG(&wrect2,vr-vs->vwinrow1,vc2-vs->vwincol1,
                vr-vs->vwinrow1,wrect.c2);
            Wfill(wnum,&wrect2,' ',A_NORM);
        }
    }
    return(0);
}
```

Najciekawszy fragment kodu funkcji **redraw** obejmuje badanie wzajemnych zależności między prostokątami w ekranie wirtualnym, oknie i ekranie fizycznym. Zgodnie z wymogami sprecyzowanymi w p. 4.4 parametr **wrectp** określa położenie odtwarzanego prostokąta w ramach

okna o numerze `wnum`. Na początku następuje normalizacja prostokąta wyznaczanego przez `wrectp`. Jest on docinany do rozmiarów okna. Uzyskuje się je badając `wrectp`, sytuujący to okno na ekranie fizycznym. Zmienne `vrow1`, `vrow2`, `vcol1`, `vcol2` wyznaczają prostokąt do odtworzenia. Jest to wykonywane wiersz po wierszu za pomocą `Wwrtcells`. Wywołania funkcji `Wfill` są konieczne, bowiem jest możliwe, że okno ma większą liczbę wierszy i/lub kolumn niż ekran wirtualny. Ten niewykorzystany fragment okna jest wypełniany spacjami.

Przykłady ilustrujące działanie funkcji z poziomu ekranów wirtualnych podano w p. 4.6.

4.6. Funkcje użytkowe

Celem tego podrozdziału jest omówienie zasad korzystania z ekranów wirtualnych i przedstawienie kilku funkcji wykonujących często spotykane zadania. Pokazują one, w jaki sposób można wykorzystać funkcje warstwy ekranów wirtualnych. Zamieszczony tu zestaw nie jest kompletny, mamy jednak nadzieję, że po zapoznaniu się z poniższymi przykładami Czytelnik będzie go mógł bez trudności samodzielnie rozszerzyć.

Pierwszym problemem, jaki należy rozstrzygnąć podczas tworzenia nowego ekranu, jest ustalenie jego wielkości. W jej określaniu należy kierować się dwoma przeciwstawnymi dążeniami: oszczędnością pamięci i prostotą kodu funkcji. To drugie wymaganie sprowadza się do unikania, w miarę możliwości, przemieszczania danych przez funkcję użytkową. Powoduje to zwiększenie skomplikowania oprogramowania w sytuacji, gdy dysponujemy dość rozbudowanymi możliwościami przemieszczania zawartości ekranu za pomocą funkcji warstwy ekranów wirtualnych.

Jeżeli dane można zapisać na ekranie o stałych i stosunkowo niewielkich rozmiarach, np. 100 wierszy i 100 kolumn, a także dysponujemy wymaganą wielkością pamięci na bufor ekranu (w tym wypadku wymaganych jest $100 \times 100 \times 2 = 20000$ bajtów), to najwygodniej jest zapisać wszystkie dane do jednego ekranu i udostępniać jego fragmenty w oknie, np. przemieszczając kursor, co jest wykonywane standardowo.

Rozwiązanie takie jednak zawiedzie, kiedy liczba wierszy jest bardzo duża, lub też nie można z góry jej ograniczyć, co występuje np. w przypadku funkcji do przeglądania zawartości pliku. Zalecane jest wówczas przyjęcie liczby wierszy ekranu równej liczbie wykorzystywanych wierszy w oknie. Całość przemieszczania pionowego zachodzi wówczas przez własną funkcję. Dzielenie odpowiedzialności za

przemieszczanie między funkcje standardowe a własne prowadzi jedynie tylko do niepotrzebnego komplikowania kodu. Analogiczna sytuacja występuje, gdy długość wiersza jest zbyt wielka lub nieokreślona.

Przedstawione tu funkcje to: *VSstart*, *VSzmien_okno*, *VSmenu* i *VSpokaz_linie*. Deklaracje tych funkcji, wraz z definicjami wykorzystywanych stałych, są zapisane w pliku nagłówkowym *vst.h*:

```
void VSstart();
void VSzmien_okno(int vs);
int VSmenu(char *txty[], char *nagl, int wybor, RECT *r);
int VSpokaz_linie(char *nagl, char *txt, RECT *r1);

#define BLAD -1
#define K_esc -2
#define K_ret -3
```

Zadaniem funkcji *VSstart* jest określenie atrybutów stosowanych do wyświetlania znaków. Atrybuty te zapisuje się w zmiennych o nazwach określających funkcje pełnione przez tekst zapisany z ich wykorzystaniem, w tym przypadku są nimi: *normalny* i *wybrany*. Przechowywanie atrybutów w zmiennych pozwala na dostosowanie sposobu wyświetlania do aktualnie używanej kombinacji karta/monitor a także na ich zmianę przez użytkownika w trakcie pracy programu.

```
static char normalny, wybrany;
```

Jeżeli zachodzi taka potrzeba, to można naturalnie wprowadzić więcej takich zmiennych. *VSstart* określa, że dla kart kolorowych tekst *normalny* tworzą żółte znaki na niebieskim tle (wartość atrybutu 26) zaś tekst *wybrany* tworzą brązowe znaki na jasnoniebieskim tle (wartość atrybutu 54). Dla kart monochromatycznych tekst *normalny* jest wypisywany z atrybutem *A_NORM*, natomiast tekst *wybrany* z *A_INVERSE*. W niektórych przypadkach określenie kolorów na podstawie stosowanej karty nie daje najlepszych efektów. Występuje to wtedy, gdy kolorowa karta jest stosowana w połączeniu z monitorem monochromatycznym. W takiej sytuacji użytkownik musi przekazać informację o rodzaju posiadanego monitora.

```
void VSstart() /* inicjowanie pracy warstwy funkcji użytkowych */
{
    BOOLEAN mono;
    BEequip(NULL, NULL, NULL, NULL, &mono, NULL);
    if (mono)
    {
        normalny=A_NORM;
        wybrany=A_INVERSE;
```

```

    }
else
    {
        normalny=BLUE_YELLOW;
        wybrany=LIGHTBLUE_BROWN;
    }
V$begin();
}

```

Funkcja **B\$equip** jest umieszczona w module bios.c. Ustala ona (za pomocą przerwania systemu BIOS) konfigurację używanego mikrokomputera. Należy zwrócić uwagę na wykorzystanie wskaźników **NULL**. Stosuje się je w celu zaznaczenia, że nie interesuje nas nadanie wartości pewnym parametrom.

Funkcja **V\$pokaz_linie** pozwala na przejrzanie ciągu o dowolnej długości w oknie o podanych rozmiarach. Z założenia poprawiana jest jedna linia, zatem skrołowanie pionowe jest zbędne. W poziomie zastosowano jednak skrołowanie niestandardowe. Pozwala to na specjalny sposób wyświetlania danych: nagłówek jest widoczny niezależnie od położenia kursora. Taki sposób wyświetlania czyni ją przydatną podczas przeglądania pól rekordu. Drugą korzyścią, wynikającą z ograniczenia wielkości ekranu, jest oszczędność pamięci. W tym konkretnym przypadku ma ona mniejsze znaczenie - ekran ma tylko 1 wiersz.

```

int V$pokaz_linie(nagl, txt, r1)
    /* edycja pojedynczej linii tekstu */
char *nagl;      /* nagłówek tekstu */
char *txt;       /* przeglądany tekst */
RECT *r;         /* okno, w którym tekst jest przeglądany */
{
    int vs, key, dl_nagl, dl_txt;
    RECT r;
    char *akt_znak, *pocz_znak;
    dl_nagl=strlen(nagl);
    dl_txt=strlen(txt);
    intersect(r1,ROWS-1,CLS-1,&r);
    if (RWIDTH(&r)<dl_nagl+1)
        return(BLAD);
    if ((vs=VSnew(1,RWIDTH(&r),&r,NULL,A_NORM))==BADVSCREEN)
        return(BLAD);
    Wsetfrm(V$getwnum(vs),F_UNFRAMED,A_NORM,NULL);
    V$write(vs,0,0,GIANT,nagl,wybrany);
    V$write(vs,0,dl_nagl,GIANT+1,txt,normalny);
}

```



```

VSsetcur(vs,0,dl_nagl);
VSshowcur(vs,TRUE);
PSsynch();
pocz_znak=akt_znak=txt;
while ((key=keycode())!=K_ESC && key!=K_RET)
{
    switch (key)
    {
        case K_LEFT:
            akt_znak--;
            break;
        case K_RIGHT:
            akt_znak++;
            break;
        case K_END:
            akt_znak=txt+dl_txt-1;
            break;
        case K_HOME:
            akt_znak=txt;
            break;
    }
    pokaz_norma(&akt_znak, &pocz_znak, txt, RWIDTH(&r)-dl_nagl,
                dl_txt);
    VSwrite(vs,0,dl_nagl,GIANT+1,pocz_znak,normalny);
    VSsetcur(vs,0,dl_nagl+(akt_znak-pocz_znak));
    PSsynch();
}
VSdispose(vs);
PSsynch();
return (key==K_RET)? K_ret: K_esc;
}

```

Na początku funkcja zapamiętuje w zmiennych lokalnych długości nagłówka i tekstu, docina przekazany prostokąt do wymiarów ekranu i sprawdza, czy jest on wystarczająco szeroki, by wyświetlić choćby fragment tekstu. Każdy nowo tworzony ekran wirtualny otrzymuje okno opatrzone ramką typu *F_SINGLE*, tj. utworzona z pojedynczej linii. W tym przypadku konieczna jest zmiana rodzaju ramki - okno to składa się z 1 wiersza i nie ma w nim miejsca na ramkę. Wywołanie funkcji *Wsetfrm* powoduje, że podczas wyświetlania tego okna system nie będzie tworzył ramki. Parametr określający długość ciągu podczas wypisywania treści

ciągu ma wartość `GIANT+1`. Pozwala to na dopisanie spacji, jeżeli długość ciągu jest mniejsza niż liczba dostępnych kolumn okna (patrz p. 4.5).

Wywołanie `VSsethrowcur` jest konieczne, bowiem bezpośrednio po utworzeniu ekranu kursor jest niewidoczny. `VSpokaz_linie` nie wykonuje przemieszczania za pomocą `VSlide`. W miejsce tego wypisywany jest zawsze cały ciąg, począwszy od `pocz_znak`. W przypadku okien jednowierszowych jest to lepsze rozwiązanie.

Funkcja `pokaz_norma` wykorzystuje dwie zmienne: `akt_znak` - wskazująca na znak, na którym spoczywa kursor oraz `pocz_znak` - wskaźnik na pierwszy wypisywany znak tekstu. Modyfikacja wartości `akt_znak` jest, jak widać, bardzo prosta. Funkcja `pokaz_norma` poprawia, jeżeli jest to konieczne, wartość `akt_znak` i dostosowuje do niej wartość `pocz_znak`:

```
static void pokaz_norma(char **akt_znak, char **pocz_znak,
                        char *txt, int dl_okna, int dl_txt)
{
    if (*akt_znak<txt)
    {
        *pocz_znak=*akt_znak=txt;
        return;
    }
    if ((*akt_znak-txt)>=dl_txt)
    {
        *akt_znak=txt+dl_txt-1;
        *pocz_znak=(*akt_znak)-dl_okna+1;
        if (*pocz_znak<txt)
            *pocz_znak=txt;
        return;
    }
    if (((*akt_znak)-dl_okna)>=*pocz_znak)
    {
        *pocz_znak=*akt_znak-dl_okna+1;
        return;
    }
    if (*akt_znak<*pocz_znak)
        *pocz_znak=*akt_znak;
}
```

Funkcja `VSzmien_okno` pozwala użytkownikowi na zmianę położenia i rozmiarów okna przypisanego do ekranu wirtualnego:

```

void VSzmien_okno() /*zmiana rozmiarów i położenia okna wirtualnego */
int vs;             /* numer ekranu wirtualnego */
{
    RECT zapas, r;
    int w, key;
    if ((Wiszoomed(w=VSgetwnum(vs)))==TRUE)
        return;
    Wgetphys(w,&r,TRUE);
    memmove(&zapas,&r,sizeof(r));
    while ((key=keycode())!=K_ESC && key!=K_RET)
    {
        switch (key)
        {
            case K_UP:
                if (r.r1>0)
                    r.r1--, r.r2--;
                break;
            ...
            case K_RIGHT:
                if (r.c2<COLS-1)
                    r.c1++, r.c2++;
                break;
            case '8':
                if (r.r2>r.r1+1)
                    r.r2--;
                break;
            ....
            case '6':
                if (r.c2<COLS-1)
                    r.c2++;
                break;
        }
        Wsetphys(w,&r);
        PSsynch();
    }
    if (key!=K_RET)
    {
        Wsetphys(w,&zapas);
        PSsynch();
    }
}

```

Pominięte fragmenty kodu realizują przemieszczanie i zmianę położenia okna w pozostałych kierunkach. Sprawdzanie poprawności rozmiarów okna musi być wykonane przez *VSzmien_okno*. Funkcja *Wsetphys* ignoruje bowiem próby utworzenia okien nie mieszczących się całkowicie na ekranie fizycznym.

Ostatnią z funkcji jest funkcja *VSmenu*. Liczba opcji wybieranych z menu powinna być stosunkowo niewielka, zatem zasadne jest przyjęcie założenia, że dane mogą być w całości zapisane w ekranie wirtualnym. Przemieszczanie odbywa się całkowicie za pomocą standardowych funkcji. Funkcja ta zwraca numer kolejny wybranego wiersza bądź też typ błędu: zaniechanie wyboru lub niemożność utworzenia ekranu wirtualnego.

```
int VSmenu(char *txty[], char *nagl, int wybor, RECT *r)
{
    int vs, l_w, l_kol, k, w, i, key;
    ustal_rozmiary(txty, &l_w, &l_kol);
    wybor=MIN(wybor, l_w-1);
    wybor=MAX(0, wybor);
    if ((vs=VSnew(l_w, l_kol, r, nagl, A_NORM))==BADVSCREEN)
        return(BLAD);
    for (i=0; i<l_w; ++i)
        VSwrite(vs, i, 0, GIANT+1, txty[i], (i==wybor)? wybrany: normalny);
    VSsetcur(vs, wybor, 0);
    PSsynch();
    while ((key=keycode())!=K_ESC && key!=K_RET)
    {
        VSgetcur(vs, &w, &k);
        wybor=w;
        switch(key)
        {
            case K_F4:
                VSzmien_okno(vs);
                break;
            case K_UP:
                w= (w==0)? l_w-1: w-1;
                break;
            case K_DOWN:
                w= (w==l_w-1)? 0: w+1;
                break;
            case K_LEFT:
                VSpan(VSgetwnum(vs), DIR_LEFT, 1);
                k--;
```

```

        break;
    case K_RIGHT:
        VSpan(VSgetwnum(vs), DIR_RIGHT, 1);
        k++;
        break;
    default:
        szukaj_litery(txt, key, &w);
        break;
    }
    if (wybor != w)
    {
        VSwrite(vs, wybor, 0, GIANT+1, txt[wybor], normalny);
        VSwrite(vs, w, 0, GIANT+1, txt[w], wybrany);
    }
    VSsetcur(vs, w, k);
    PSSynch();
}
VSdispose(vs);
PSSynch();
return (key == K_RET)? w: K_esc;
}

```

Rozmiar ekranu wirtualnego określa funkcja `ustal_rozmiary`, wyznaczająca liczbę ciągów oraz ich maksymalną długość. Wszystkie one są wpisywane do ekranu z właściwymi atrybutami. Wybrany wiersz to ten, na którym spoczywa (niewyświetlany) kursor. Zmianę wyświetlanego fragmentu ekranu uzyskuje się w pionie przemieszczaniem kursora, zaś w poziomie wywołaniem `VSpan`. Stosowanie `VSpan` jest łatwiejsze niż przesuwanie kursora. Stosowanie tej drugiej metody wymaga określenia położenia okna w ramach ekranu przed wywołaniem `VSsetcur`.

Użytkownik ma poza tym możliwość zmiany rozmiarów okna (wywołanie funkcji `VSzmien_okno` po naciśnięciu F4) oraz automatycznego znajdowania pierwszego wiersza, zawierającego wprowadzaną z klawiatury dużą literę alfabetu – funkcja `szukaj_litery`:

```

static void szukaj_litery(char *txt[], int key, int *w)
{
    char **linie, *txt;
    linie=txt;
    while (**linie && *linie!=NULL)
    {
        txt=*linie;
        while (*txt)

```

```
    if (*txt++==toupper(key))
    {
        *w=linie-txt;
        return;
    }
    *(linie++);
}
}
```

Rozdział 5

KLAWIATURA

Rozdział ten omawia pakiet funkcji służących do obsługi klawiatury. Podobnie, jak i rozdział poprzedni dotyczący ekranu, opracowany został w głównej mierze na podstawie pracy [ADV]. Również i tutaj nacisk jest położony na zapewnienie przenośności oprogramowania.

Często nie zdajemy sobie sprawy, w jak wielkim stopniu komfort pracy użytkownika zależy od poprawnego zaprojektowania modułu obsługi klawiatury. Początkujący programiści mają problemy z wykonaniem podstawowych operacji, takich jak: poruszanie kursora znakami strzałek, lub też sprawdzanie czy uwzględniono wszystkie wprowadzone przez użytkownika znaki. Obsługa klawiatury obejmuje jednak znacznie bardziej skomplikowane operacje, takie jak np. definiowanie makr lub zmiana przez użytkownika sposobu interpretacji klawiszy bez potrzeby ingerencji w kod źródłowy. Wszystkie te, jak i inne pokrewne im problemy, zostaną omówione w dalszej części rozdziału. W wielu przypadkach zaprezentowano kod realizujący odpowiednie funkcje.

Obsługa klawiatury odbywa się w trzech warstwach: interfejs, klawiatura wirtualna i funkcje użytkowe. Podział ten wyznacza strukturę rozdziału. Przedstawiona implementacja jest przeznaczona dla systemu operacyjnego MS DOS. Adaptacja tych funkcji na potrzeby systemu Unix nie powinna sprawić większych problemów, więcej informacji na ten temat znaleźć można w pracy [ADV].

5.1. Podstawowe pojęcia

W systemie MS DOS istnieje wiele możliwości pobrania znaku z klawiatury. Można używać do tego celu funkcji systemowych, przerwać systemu BIOS lub sterować klawiaturą bezpośrednio przez bufor systemowy i odpowiednie porty. Ponieważ większość komputerów PC/AT jest kompatybilna na poziomie systemu BIOS, wobec tego wybór tego poziomu wydaje się najodpowiedniejszy. Dla klawiatury, w przeciwieństwie do obsługi ekranu za pomocą systemu BIOS, poziom ten zapewnia wystarczającą szybkość działania. Zmiana metody obsługi

klawiatury (wybór innego sposobu, odpowiedniejszego dla danego komputera) nie wymaga przeprogramowywania całego modułu, lecz tylko korekty najniższej warstwy, tzn. interfejsu. Zmiany muszą być wprowadzone do trzech, opisanych poniżej dokładnie funkcji.

Do pliku bios.c, gdzie mieszczą się już uprzednio omówione funkcje wykorzystujące system BIOS, dodajemy trzy nowe funkcje dotyczące klawiatury: **BKavail**, **BKread** i **BKstatus**. Pierwsza z nich sprawdza zawartość systemowego bufora klawiatury. Jeżeli jest w buforze choć jeden niepobraný znak, to funkcja ta przyjmuje wartość **TRUE**.

```
BOOLEAN BKavail();
```

Następna z nich, **BKread**, udostępnia kod naciśniętego klawisza (tzw. scan code) i kod znaku z tablicy ASCII lub 0:

```
int BKread(int *scanp)
```

Dla znaków ASCII wynikiem funkcji jest kod wciśniętego znaku. Dla klawiszy specjalnych na klawiaturze IBM PC (takich jak F1, PgUp, Del) wynikiem funkcji **BKread** jest 0. Znaki te są rozróżniane przez wartość nadawaną parametrowi scanp. Funkcja ta pozwala także na rozpoznanie zdublowanych na klawiaturze numerycznej znaków, takich jak plus, minus i gwiazdka.

Ostatnia z funkcji, **BKstatus**, zwraca liczbę, której mniej znaczący bajt jest ustawiony zgodnie ze schematem:

```
bit 7 klawisz Insert włączony
    6 Caps Lock włączony
    5 Num Lock włączony
    4 Scroll Lock włączony
    3 Alt wciśnięty
    2 Ctrl wciśnięty
    1 lewy Shift wciśnięty
    0 prawy Shift wciśnięty
```

Trzy opisane funkcje stanowią podstawę, na jakiej zostanie zbudowany prosty interfejs komunikacji między klawiaturą a użytkownikiem. Tworzą ją dwie funkcje: **cget** i **cready**. Pierwszą z wprowadzanych funkcji jest funkcja **cget**. Jej wynikiem jest kod naciśniętego znaku. Gdy wynikiem jest wartość 0, wówczas do wczytania całego kodu wciśniętego klawisza konieczne jest jej powtórne wywołanie:

```
static nextch=0;
int cget(void)
{
    int ch;
```



```

if (nextch != 0)
{
    ch = nextch;
    nextch = 0;
}
else
{
    ch = BKread(&nextch);
    if (ch != 0)
        nextch = 0;
}
return(ch);
}

```

Po zwróceniu przez **BKread** wartości 0, zapamiętany zostaje scan kod w lokalnej zmiennej zewnętrznej *nextch*. Jest on wynikiem przy powtórnym wywołaniu funkcji *cget*.

Następna pomocnicza funkcja to **cready**, która sprawdza, czy jakiś znak z klawiatury znajduje się już w buforze i jest gotowy do pobrania. Dotyczy to zarówno bufora systemowego, jak i zmiennej *nextch*. Funkcja ta nie usuwa tego znaku z bufora.

```

BOOLEAN cready(void)
{
    BOOLEAN BKavail();
    return((BOOLEAN)(nextch != 0 || BKavail()));
}

```

5.2. Interfejs dla klawiatury

Najniższa warstwa obsługi klawiatury zawiera dodatkowo funkcje **setraw** i **unsetraw**. Są one konieczne tylko w systemach, w których nieodzwonne jest sterowanie terminalem (np. przełączenie go w tryb znakowy pracy, określenie sposobu wczytywania znaków). W systemie MS DOS są one zbędne, zostawia się je w celu zachowania przenośności kodu:

```

void setraw(void);    /* inicjowanie klawiatury */
void unsetraw(void); /* zakończenie pracy z klawiaturą */

```

Inicjacja modułu klawiatury i jej epilog następuje po wywołaniu **Kbegin** i **Kend**, w których to funkcjach są wywoływane odpowiednio **setraw** i **unsetraw**:

```
void Kbegin(void)      /* początek pracy z klawiaturą */
void Kend(void)        /* koniec pracy z klawiaturą */
```

Do pobrania znaku używa się **Kget**, w której następuje wywołanie **keycode**. Funkcja **keycode** składa się tutaj jedynie z wywołanie **cget**, jednak – jak się to później okaże – czasami celowe jest jej znaczne rozbudowanie.

```
int Kget(void)         /* pobranie kodu z klawiatury */
{
return(keycode());
}

int keycode(void)
{
return(cget());
}
```

Do sprawdzenia, czy dostępny jest jakikolwiek znak, służy funkcja **Kready**:

```
BOOLEAN Kready(void)  /* czy jest znak do pobrania */
{
return(cready());
}
```

Na koniec wprowadzamy funkcję **Ksynch**. Stosowanie jej w miejsce **PSsynch** pozwala na zaniechanie aktualizacji ekranu, jeżeli w buforze są nieprzetworzone znaki. Dzięki temu zmniejsza się liczba aktualizacji ekranu i tym samym zwiększa się szybkość działania programu:

```
void Ksynch()          /* warunkowa synchronizacja ekranu */
{
if (!Kready())
    PSsynch();
}
```

Wszystkie wymienione funkcje znajdują się w pliku **keyboard.c**.

5.3. Klawiatura wirtualna

Zdefiniowany interfejs nie rozwiązuje jednak wszystkich problemów. Jako przykład niech posłuży eksperyment – wprowadzamy na mikrokomputerze kompatybilnym z IBM PC/AT następujący ciąg znaków: **h e l l o F1 F2**. Wypisując na ekran kolejne wartości funkcji **keycode** otrzymamy ciąg: 104 101 108 108 111 0 59 0 60. Po wprowadzeniu tych samych znaków z terminala prawdopodobnie otrzymamy coś innego,

zakładając naturalnie, że terminal ten zawiera klawisze funkcyjne F1 i F2, co nie jest bynajmniej regułą. Bardzo komplikuje to algorytm obsługujący wejściowy strumień znaków pobieranych z klawiatury. Istnieją trzy źródła tych kłopotów:

1. Pewne klawisze generują kilka znaków; czasami są to ciągi zmiennej długości; program musi wobec tego rozpoznawać, kiedy sekwencja się zaczyna a kiedy kończy.
2. Sekwencje odpowiadające tym samym znakom różnią się między różnymi typami terminali, a nawet ten sam terminal może różnie się zachowywać w zależności od tego, w jaki sposób został skonfigurowany.
3. Nie wszystkie terminale dysponują takim samym zestawem znaków.

Rozwiązaniem pierwszego problemu jest definicja klawiatury wirtualnej, w której każdej możliwej kombinacji klawiszy jest przyporządkowany niepowtarzalny kod. Bardziej złożona, tablicowa implementacja schematu klawiatury wirtualnej, pozwoli na rozwiązanie drugiego i trzeciego problemu.

Początkowy blok 128 kodów klawiatury wirtualnej jest interpretowany jak kody ASCII. Kody od 128 do 299 są nieużywane. Klawisze funkcyjne zaczynają się od kodu 300 i wzrastają w miarę potrzeby. W przedstawionym rozwiązaniu najwyższy używany kod to 396. Przyporządkowanie takie jest zdefiniowane w zbiorze nagłówkowym `keycode.h`.

Zbiór rozpoczynają znaki kontrolne i sterujące ASCII.

```
#define K_CTL_A      1      -      Ctrl-A
#define K_CTL_B      2
#define K_CTL_C      3
#define K_CTL_D      4
#define K_CTL_E      5
#define K_CTL_F      6
#define K_CTL_G      7
#define K_CTL_H      7      -      brzęczyk
#define K_CTL_I      8
#define K_CTL_J      8      -      kasowanie znaku przed kursorem
#define K_CTL_K      9
#define K_CTL_L      9      -      tabulacja
#define K_CTL_M     10
#define K_CTL_N     10      -      nowa linia
#define K_CTL_O     11
#define K_CTL_P     12
```

```

#define K_CTL_M      13
#define K_RET        13    -    return
#define K_CTL_N      14
#define K_CTL_O      15
#define K_CTL_P      16
#define K_CTL_Q      17
#define K_CTL_R      18
#define K_CTL_S      19
#define K_CTL_T      20
#define K_CTL_U      21
#define K_CTL_V      22
#define K_CTL_W      23
#define K_CTL_X      24
#define K_CTL_Y      25
#define K_CTL_Z      26
#define K_CTL_LB     27    -    Ctrl-[
#define K_ESC        27    -    Esc
#define K_CTL_BSLSH  28    -    Ctrl-\
#define K_CTL_RB     29    -    Ctrl-]
#define K_CTL_CFLEX  30    -    Ctrl-^
#define K_CTL_UNDER  31    -    Ctrl-_
#define K_DEL        127   -    kasowanie

```

Następnie są zdefiniowane kody najczęściej używane w systemach ekranowego wprowadzania danych i edytorach tekstowych. W nawiasach kwadratowych podano odpowiadające im klawisze w mikrokomputerach kompatybilnych z IBM PC/AT:

```

#define K_UP         301    -    kursor w górę      [Up arrow ]
#define K_DOWN       302    -    kursor w dół      [Down arrow ]
#define K_RIGHT      303    -    kursor w prawo   [Right arrow]
#define K_LEFT       304    -    kursor w lewo    [Left arrow]
#define K_PREV       305    -    strona poprzednia [PgUp]
#define K_NEXT       306    -    strona następna  [PgDn]
#define K_LHOME      307    -    początek wiersza [Ctrl-Home]
#define K_LEND       308    -    koniec wiersza  [Ctrl-End]
#define K_HOME       309    -    początek strony [Home]
#define K_END        310    -    koniec strony  [End]
#define K_INS        311    -    wstawianie    [Ins]
#define K_WLEFT      312    -    słowo w lewo   [Ctrl-Left arrow]
#define K_WRIGHT     313    -    słowo w prawo  [Ctrl-Right arrow]
#define K_BTAB       315    -    tabulacja w tył [Shift-Tab]
#define K_SCL_UP     321    -    okno w górę    [Ctrl-u]

```

```
#define K_SCL_DOWN 322 - okno w dół      [Ctrl-j]
#define K_SCL_RIGHT 323 - okno w prawo    [Ctrl-l]
#define K_SCL_LEFT 324 - okno w lewo     [Ctrl-k]
#define K_ZOOM_OUT 325 - pomniejsz       [klaw. numeryczna -]
#define K_ZOOM_IN 326 - powiększ        [klaw. numeryczna +]
```

Następnie zdefiniowano 10 klawiszy funkcyjnych:

K_F1-331, ..., K_F10-340 - F1, ..., F10.

Komputer klasy IBM PC dysponuje dodatkowo 30 sekwencjami tworzonymi przez jednoczesne naciśnięcie Ctrl, Shift lub Alt z klawiszem funkcyjnym F1 ... F10. Coraz częściej spotykane są też klawiatury wyposażone w 12 klawiszy funkcyjnych. Wykorzystanie ich może przyczynić się do przyspieszenia pracy doświadczonego użytkownika systemu, ale też może doprowadzić do utrudnienia korzystania z systemu przez początkujących użytkowników, tak jak w edytorze WordPerfect. Należy też pamiętać, że klawisze te nie występują w terminalach systemu UNIX.

Sposób tworzenia tych stałych jest analogiczny do stosowanego powyżej schematu:

K_S_F1-341, ..., K_S_F10-350 - Shift-F1, ..., Shift-F10,

K_C_F1-351, ..., K_C_F10-360 - Ctrl-F1, ..., Ctrl-F10,

K_A_F1-361, ..., K_A_F10-370 - Alt-F1, ..., Alt-F10.

Ostatnie 26 kodów obsługuje litery alfabetu używane z jednoczesnym naciśnięciem klawisza Alt, czyli:

K_ALT_A-371, ..., K_ALT_Z-396 - Alt-A, ..., Alt-Z.

Wszystkie kody o wartościach powyżej 127 muszą być identyfikowane przez ich symboliczne nazwy, nigdy natomiast przez faktyczny numeru kodu czy bezpośrednio przez sekwencję wysyłaną przez daną klawiaturę. To upraszcza znacznie obsługę, zwiększa czytelność kodu źródłowego i ułatwia przenośność oprogramowania na inne komputery.

Sposób odwzorowania wczytanej sekwencji na kod wirtualny zostanie wyjaśniony w dwóch następnych rozdziałach.

5.3.1. Stała mapa wirtualna

W przeciwieństwie do komputerów kompatybilnych z IBM PC/AT, terminale nie mają niektórych klawiszy (brak jest np. ALT i klawiszy funkcyjnych F1 ... F12), co zmniejsza liczbę kombinacji dostępnych dla użytkownika. Jednakże i tę trudność można pokonać. Poniższy rozdział omawia jedno z możliwych rozwiązań. Polega on na wykorzystaniu klawisza Ctrl, dostępnego praktycznie w każdym terminalu.

Klawiatura terminala pozwala na uzyskanie 32 kodów powstałych przy równoczesnym naciśnięciu dwóch klawiszy, z których jednym jest Ctrl. W uzupełnieniu do 26 kodów reprezentowanych przez Ctrl-A do Ctrl-Z występują jeszcze Ctrl-@, Ctrl-[(znany jako Esc), Ctrl-, Ctrl-], Ctrl-^ i Ctrl-_. W praktyce jednak zalecane jest nieuwzględnianie tych ostatnich kombinacji klawiszy - wielu użytkowników, szczególnie mających tylko okazjonalny kontakt z klawiaturą, ma trudności z ich zapamiętaniem. Wykluczyć należy także kombinacje mające standardową interpretację: Ctrl-H (kasowanie znaku przed kursorem), Ctrl-I (tabulacja) i Ctrl-M (nowa linia). Do dyspozycji zostały zatem 23 kombinacje, które można wykorzystać do wygenerowania 59 kodów funkcyjnych.

Jak widać, w terminalach nie istnieje wystarczająca liczba klawiszy. W proponowanym rozwiązaniu posługujemy się kombinacją Ctrl-Q jako kodem początkowym dla sekwencji dwuznakowej. Przyporządkowanie jest następujące:

K_ALT_A reprezentuje sekwencję Ctrl-Q Ctrl-A,
 K_ALT_B - !! - - !! - Ctrl-Q Ctrl-B itd.
 Podobnie dla K_F1 do K_F10:
 K_F1 reprezentuje sekwencję Ctrl-Q 1,
 k_F10 - !! - - !! - Ctrl-Q 0.

Przedstawiona poniżej wersja funkcji **keycode** pokazuje jak można wykorzystać powyższe ustalenia do obsługi edytora na terminalu w sposób zbliżony do popularnego edytora tekstów Wordstar. Funkcja **keycode** przekształca rzeczywiste kody zwracane przez cget na kody wirtualne. Występująca w niej instrukcja switch obsługuje pojedyncze kody znaków (takie jak Ctrl-A), natomiast zagnieżdżona w niej druga instrukcja switch zajmuje się sekwencjami zaczynającymi się od Ctrl-Q. Oto jej implementacja:

```
#include <display.h>
#include <keycode.h>
#include <pscreen.h>
int keycode()
{
  int ch;
  while(TRUE)
    switch(ch = cget())
    {
      case 0:
        (void) cget(); /* dla klawiszy specjalnych */
        PSbeep();
```

```

        continue;
case K_CTL_A:
    return(K_WLEFT);
case K_CTL_B:
    return(K_BTAB);
case K_CTL_C:
    return(K_NEXT);
case K_CTL_D:
    return(K_RIGHT);
case K_CTL_E:
    return(K_UP);
case K_CTL_F:
    return(K_WRIGHT);
case K_CTL_J:
    return(K_SCL_DOWN);
case K_CTL_K:
    return(K_SCL_LEFT);
case K_CTL_L:
    return(K_SCL_RIGHT);
case K_CTL_O:
    return(K_ZOOM_OUT);
case K_CTL_P:
    return(K_ZOOM_IN);
case K_CTL_Q:
    switch(cget())
    {
        case '1':
            return(K_F1);
            *****
        case '9':
            return(K_F9);
        case '0':
            return(K_F10);
        case K_CTL_A:
            return(K_ALT_A);
            *****
        case K_CTL_Z:
            return(K_ALT_Z);
        default:
            PSbeep();
            continue;
    }
}

```

```

case K_CTL_R:
    return(K_PREV);
case K_CTL_S:
    return(K_LEFT);
case K_CTL_T:
    return(K_LHOME);
case K_CTL_U:
    return(K_SCL_UP);
case K_CTL_V:
    return(K_INS);
case K_CTL_W:
    return(K_HOME);
case K_CTL_X:
    return(K_DOWN);
case K_CTL_Y:
    return(K_LEND);
case K_CTL_Z:
    return(K_END);
case K_CTL_BSLSH: /* wyjście bezpieczeństwa */
    cleanup();
    exit(0);
case K_CTL_CFLEX: /* znak niekodowany */
    return(cget());
default:
    return(ch);
}
}

```

Można zauważyć, że sekwencje zaczynające się zerem są ignorowane. Znak Ctrl-^ powoduje, że następny znak nie jest tłumaczony według schematu podanego w funkcji `keycode`, co może być czasami przydatne. Aby otrzymać na wyjściu Ctrl-A, należy nacisnąć Ctrl-^ Ctrl-A.

Mając do dyspozycji tak zdefiniowaną funkcję `keycode`, zbiory nagłówkowe `keycode.h` i `keyboard.h` (zawierające deklaracje funkcji modułu klawiatury) po wprowadzeniu ciągu:

```
h e l l o Ctrl-E Ctrl-X Ctrl-Q-1 Ctrl-Q-2
```

na wyjściu funkcji `Kget` otrzymamy ciąg:

```
104 101 108 108 111 K_UP K_DOWN K_F1 K_F2.
```


Lepszym i znacznie łatwiejszym rozwiązaniem od przedstawionej mapy kodowej jest tablica kodowa wraz z procedurą przeszukującą. Poniżej zostanie zaprezentowana taka implementacja funkcji `keycode`, tym razem przystosowana dla klawiatury IBM PC. Kody w tablicy są kodami zwracanymi jako drugi znak sekwencji rozpoczynającej się zerem:

```
#include <display.h>
#include <keycode.h>
#include <pscreen.h>

#define MINCODE 15      /* K_BTAB */
#define MAXCODE 119    /* K_LHòME */
#define UNUSED 0
#define MAX_KB 200

static int key_buff[MAX_KB];
static int kmap[]=
(
    K_BTAB,   K_ALT_Q,  K_ALT_W,  K_ALT_E,
    K_ALT_R,  K_ALT_T,  K_ALT_Y,  K_ALT_U,
    K_ALT_I,  K_ALT_O,  K_ALT_P,  UNUSED,
    UNUSED,   UNUSED,   UNUSED,   K_ALT_A,
    K_ALT_S,  K_ALT_D,  K_ALT_F,  K_ALT_G,
    K_ALT_H,  K_ALT_J,  K_ALT_K,  K_ALT_L,
    UNUSED,   UNUSED,   UNUSED,   UNUSED,
    UNUSED,   K_ALT_Z,  K_ALT_X,  K_ALT_C,
    K_ALT_V,  K_ALT_B,  K_ALT_N,  K_ALT_M,
    UNUSED,   UNUSED,   UNUSED,   UNUSED,
    UNUSED,   UNUSED,   UNUSED,   UNUSED,
    K_F1,     K_F2,     K_F3,     K_F4,
    K_F5,     K_F6,     K_F7,     K_F8,
    K_F9,     K_F10,    UNUSED,   UNUSED,
    K_HOME,   K_UP,     K_PREV,   K_ZOOM_OUT,
    K_LEFT,   UNUSED,   K_RIGHT,  K_ZOOM_IN,
    K_END,    K_DOWN,   K_NEXT,   K_INS,
    K_DEL,    K_S_F1,   K_S_F2,   K_S_F3,
    K_S_F4,   K_S_F5,   K_S_F6,   K_S_F7,
    K_S_F8,   K_S_F9,   K_S_F10,  K_C_F1,
    K_C_F2,   K_C_F3,   K_C_F4,   K_C_F5,
    K_C_F6,   K_C_F7,   K_C_F8,   K_C_F9,
    K_C_F10,  K_A_F1,   K_A_F2,   K_A_F3,
    K_A_F4,   K_A_F5,   K_A_F6,   K_A_F7,
    K_A_F8,   K_A_F9,   K_A_F10,  UNUSED,
```

```
K_WLEFT, K_WRIGHT, K_LEND,  UNUSED,
K_LHOME,
};

int keycode(void)
{
    int ch, code;
    while(TRUE)
        switch(ch=cget())
        {
            case 0:
                ch=cget();
                if (ch>MINCODE && ch<=MAXCODE)
                {
                    code=kmap[ch-MINCODE];
                    if (code!=UNUSED)
                        return(code);
                }
                PSbeep();
                continue;
            case K_CTL_J:
                return(K_SCL_DOWN);
            case K_CTL_K:
                return(K_SCL_LEFT);
            case K_CTL_L:
                return(K_SCL_RIGHT);
            case K_CTL_U:
                return(K_SCL_UP);
            case K_CTL_CFLEX:
                return(cget());
            default:
                return(ch);
        }
}
```

Pozostawiono cztery kody kontrolne, sterujące przesuwaniem okna w poziomie i pionie, jak też opisany już mechanizm Ctrl-^.

5.3.2. Zewnętrzna mapa wirtualna

Stała mapa wirtualna pozwala na obsługę dowolnego terminala. Jednakże po zmianie typu terminala prawdopodobnie konieczna będzie modyfikacja kodu źródłowego w celu wygenerowania nowej wersji programu. Oznacza to, że nie możemy mieć jednej wersji programu pracującego w

sieci, do której są podłączone różnego typu terminale. Dlatego też dużo lepszym, ale i niewątpliwie bardziej skomplikowanym rozwiązaniem, jest zastosowanie zewnętrznej tablicy definicyjnej. Jest ona umieszczona w pliku kbdef.tbl i określa kody wirtualne, odpowiadające sekwencjom wcisniętych klawiszy.

Wirtualne kody są liczbami naturalnymi i znajdują się w keycode.h. Sekwencje wejściowych klawiszy mogą być złożone z dowolnych znaków, jakimi dysponuje dany terminal. W trakcie inicjowania pracy systemu plik kbdef.tbl jest wczytywany i zawarte w nim dane są organizowane w taki sposób, by móc rozpoznać każdą zdefiniowaną sekwencję, a w przypadku wprowadzenia sekwencji niezdefiniowanej – wysłać sygnał o błędzie. Zadanie to wykonuje automat skończony.

Można więc podzielić to zadanie na trzy etapy:

- projektowanie tablicy definicyjnej
- konstrukcja automatu rozpoznającego (kompilacja tablicy)
- interpretacja sekwencji wejściowych klawiszy.

5.3.2.1. Tablica definicyjna dla klawiatury

Każdy wiersz tablicy definicyjnej składa się z wirtualnego kodu sekwencji klawiszy, po którym występuje odpowiadająca mu sekwencja wejściowa. Oba te składniki może dzielić dowolna liczba spacji i tabulacji. Sekwencją wejściową jest ciąg kodów klawiszy, jakie należy wcisnąć, aby uzyskać kod wirtualny. Tworzy go ciąg liczb rozdzielanych znakami spacji lub tabulacji, np. kod wirtualny oznaczający ruch kursora do góry mógłby wyglądać następująco:

301 27 91 65 (Esc [A).

Zapis ten jest jednak mało czytelny. W celu uproszczenia kodowania dowolnej sekwencji wprowadzono następujące reguły:

1. Pojedynczy znak z wyjątkiem spacji, tabulacji i # (hash) nie zmienia swojego znaczenia.
2. Dwuznakowy symbol rozpoczynający się od ^ zastępuje znak kontrolny, np. ^A oznacza Ctrl-A.
3. Sekwencja dwóch lub więcej cyfr, nie rozpoczynająca się od 0, jest interpretowana jako liczba naturalna. Początkowe 0 powoduje, że ciąg jest interpretowany jako liczba ósemkowa.

4. Liczby mogą być poprzedzone znakiem '+'. Jego występowanie powoduje, że ciąg jest interpretowany jako liczba, niezależnie od jego długości. Pozwala to na łatwe wpisywanie liczb w systemie dziesiętnym o wartości mniejszej od 10. Bez tej reguły byłyby one interpretowane jako znaki zgodnie z pierwszą z wymienionych reguł.
5. Znak # (hash) rozpoczyna komentarz, jego drugim ogranicznikiem jest koniec wiersza.
6. Znaki spacji, tabulacji i # (hash) mogą być kodowane tylko jako liczby naturalne 32, +9 i 35.
7. Znaki wejściowe, które są również swym własnym kodem wirtualnym (np. A, =), nie muszą być kodowane w tablicy. Wynika to stąd, że jeżeli funkcja *keycode* wykryje nie rozpoznany znak na początku sekwencji, to jest on po prostu przekazywany bez dokonywania konwersji. Wystąpienie niedopuszczalnego znaku w środku sekwencji powoduje sygnalizację błędu.

Tablica definiująca klawiaturę typu Wordstar dla IBM PC/AT jest zamieszczona w załączniku.

5.3.2.2. Konstrukcja automatu skończonego

Jak już poprzednio wspomniano, przed przystąpieniem do rozpoznawania sekwencji wejściowych konieczne jest skonstruowanie - na podstawie pliku *kbdef.tbl* - automatu skończonego. Sposób jego budowy najlepiej można prześledzić na przykładzie analizy fragmentu z tablicy:

```
301 27 [ A
302 27 [ B
303 [ C
304 [ D
305 27 ) A
306 27 ) B 2
```

Algorytm postępowania jest następujący:

1. Opuść puste wiersze i komentarze.
2. Przekształć wszystkie pozostałe składniki tablicy na liczby naturalne, zgodnie z wymienionymi regułami. W wyniku tego otrzymujemy tablicę kodów wirtualnych w postaci liczb naturalnych i powiązane z nimi sekwencje wejściowe, także zapisane w postaci ciągów liczb naturalnych:

301: 27 91 65
302: 27 91 66
303: 91 67
304: 91 68
305: 27 41 65
306: 27 41 66 50

3. Posortuj tablice rosnaco według sekwencji wejściowych.
Otrzymujemy wówczas:

305: 27 41 65
306: 27 41 66 50
301: 27 91 65
302: 27 91 66
303: 91 67
304: 91 68

4. Podziel tablicę na podtablice według pierwszych liczb w każdej sekwencji. Każda taka podtablica jest nowym stanem i jest oznaczana unikatowym numerem.

Liczba 27, stan 1:

305: (27) 41 65
306: (27) 41 66 50
301: (27) 91 65
302: (27) 91 66

Liczba 91, stan 2:

303: (91) 67
304: (91) 68

Niech termin i -ta podtablica oznacza taką podtablicę, w której co najmniej i pierwszych znaków jest identycznych (w przykładach są one wyróżnione umieszczeniem ich w nawiasach). Oznacza to, że właśnie utworzona została pierwsza podtablica. Wejściową tablicę można zatem traktować jako podtablicę zerową.

5. Podziel według powyższej zasady każdą z podtablic; otrzymasz:

Liczba 27, stan 1:

Liczba 41, stan 1.1:

305: (27 41) 65
306: (27 41) 66 50

Liczba 91, stan 1.2:

301: (27 91) 65
302: (27 91) 66

Liczba 91, stan 2:

Liczba 67, stan 2.1:

303: (91 67)

Liczba 68, stan 2.2

304: (91 68)

Postępując dalej w ten sam sposób dzielimy kolejne podtablice. Proces ten kończy się, gdy każdemu kodowi wirtualnemu zostanie przyporządkowany unikalny stan. Sekwencja wejściowa znajdzie się wtedy całkowicie w nawiasach. W każdej podtablicy znajduje się dokładnie jedna sekwencja wejściowa. Ostatecznie otrzymamy zatem:

Liczba 27, stan 1:

Liczba 41, stan 1.1:

Liczba 65, stan 1.1.1:

305: (27 41 65)

Liczba 50, stan 1.1.1.1:

306: (27 41 66 50)

Liczba 91, stan 1.2:

Liczba 65, stan 1.2.1:

301: (27 91 65)

Liczba 66, stan 1.2.2:

302: (27 91 66)

Liczba 91, stan 2:

Liczba 67, stan 2.1:

303: (91 67)

Liczba 68, stan 2.2

304: (91 68)

Jeżeli podczas konstruowania automatu okaże się, że:

- istnieją dwie lub więcej identyczne sekwencje wejściowe
 - pewna sekwencja wejściowa jest przedrostkiem innej sekwencji,
- to zostanie zasygnalizowany błąd.

Tak skonstruowany automat działa w sposób deterministyczny. Jeżeli podczas analizy wczytywanej przez *cget* sekwencji kodów okaże się, że nie została ona zdefiniowana, to sygnalizowany jest błąd i następuje powrót do stanu początkowego. Jedynym odstępstwem od tej zasady jest sytuacja, w której automat, znajdujący się w stanie początkowym, nie rozpozna znaku. Wówczas znak ten jest traktowany literalnie, czyli tak jak swój własny kod. Dzięki temu znaki ASCII i ich kody nie muszą być umieszczane w tablicy.

Proces wyszukiwania odpowiedniej sekwencji w tak zdefiniowanym automacie ilustruje poniższy przykład:

Załóżmy, że przeczytano dwa znaki Esc i ')'. Kodami ich są 27 oraz 41. Znajdujemy się więc w drugiej podtablicy:

305: (27 41) 65

306: (27 41) 66 50

Następny wczytany znak porównujemy z dwiema wartościami: 65 i 66. Jeśli jest to 65, to rozpoznano wirtualny kod 305. Jeśli znakiem tym jest 66, to przechodzimy do trzeciej podtablicy i czytany jest następny znak. Proces ten jest kontynuowany tak długo, aż zostanie wykryty błąd lub określony kod wirtualny.

Algorytm ten można formalnie zapisać następująco:

1. Zeruj zmienną *i*.
2. Czytaj znak.
3. Przeszukuj bieżącą *i*-tą podtablicę aż do napotkania wczytanego znaku.
4. Kontynuuj przeszukiwanie dla (*i*+1) podtablicy i następnego znaku.
5. Jeżeli *i*-ta podtablica jest pusta, to przerwij, gdyż nie ma takiej sekwencji i tylko w przypadku, jeżeli *i* jest równe zeru, to nie sygnalizuj błędu i przekaż ten znak.
6. Jeżeli *i*-ta podtablica zawiera tylko jedną sekwencję oraz *i*-ty znaleziony znak jest ostatni, to przekaż odpowiadający kod wirtualny i zakończ przeszukiwanie.
7. Inkrementuj zmienną *i*.
8. Wróć do kroku 2 i kontynuuj działanie.

Ten dość skomplikowany algorytm można przyspieszyć przez zapamiętanie minimalnej i maksymalnej wartości dla początkowych znaków sekwencji. Już na samym początku znak spoza tego zakresu powoduje przerwanie działania algorytmu i nie potrzebujemy analizować kolejnych znaków z takiej sekwencji. W przypadku IBM PC/AT działanie znacznie się upraszcza, gdyż każda sekwencja "specjalna" rozpoczyna się zerem. W większości pozostałych klawiatur sekwencje rozpoczynają kody pomiędzy 0 i 31.

Poniżej przedstawiono nową definicję funkcji **keycode**. Wykonuje ona wszystkie konieczne operacje, tj. wczytuje plik z definicjami kodów wirtualnych, tworzy automat do ich rozpoznawania i analizuje wczytywane sekwencje wejściowe.

Plik `kc_table.c` rozpoczynają definicje i deklaracje wykorzystywanych stałych, typów i zmiennych.

```

#include <display.h>
#include <dsputil.h>
#include <pscreen.h>
#include <keycode.h>

#define TBLGRANULE 10      /* liczba, o którą zwiększamy */
#define SEQGRANULE 3      /* liczba, o którą zwiększamy sekwencję */
#define MAXINT 32767      /* maksymalny kod wirtualny */
typedef struct {          /* tablica definicji klawiatury */
    short code;           /* kod wirtualny */
    char len;             /* długość sekwencji */
    char seq[SEQGRANULE]; /* sekwencja uderzeń */
} KEYSEQ;

static KEYSEQ ** table;   /* tablica definicji */
static int tablelen;      /* liczba definicji w tablicy */
static int maxtablelen;   /* maksymalna liczba definicji */
static int minchar = 255; /* minimalny kod znaku */
static int maxchar = 0;   /* maksymalny kod znaku */
static char * filename = "kbdef.tbl";
                          /* nazwa pliku z definicjami */
static int linenum = 0;   /* liczba wierszy w tablicy */
static FILE * fin;        /* strumień wejściowy */

```

Każda sekwencja wejściowa i odpowiadający jej kod wirtualny są zapamiętywane w strukturze **KEYSEQ**.

Na uwagę zasługuje przyjęty w poniższych funkcjach sposób gospodarowania pamięcią. Zarówno liczba kodów wirtualnych, jak i długości przypisanych im sekwencji wejściowych, mogą się w różnych aplikacjach znacznie różnić i dlatego na początku zmienna **table** wskazuje na obszar pamięci wystarczający dla **TBLGRANULE** kodów wirtualnych o sekwencjach nie przekraczających **SEQGRANULE** bajtów. Jeżeli okaże się, że stosowana jest większa liczba kodów, to następuje realokacja pamięci (zwiększenie obszaru tablicy) o następne **TBLGRANULE** pozycji. W ten sposób, jeżeli dysponujemy tylko wystarczającą dużą wolną stertą, wczytane zostaną wszystkie dane z pliku.

Podobnie, jeżeli pojedyncza sekwencja wejściowa jest dłuższa niż zadeklarowana tablica o długości **SEQGRANULE**, następuje realokacja struktury **KEYSEQ** o **SEQGRANULE** bajtów. Pamięć alokowana jest w większych blokach, ponieważ pozwala to na uniknięcie fragmentacji pamięci na stercie. Podczas wczytywania znacznej liczby definicji fragmentacja ta groziłaby znacznym spowolnieniem pracy systemu (patrz p. 3.3).

Zmienna **maxtablelen** zawiera maksymalną liczbę sekwencji klawiszy, natomiast **tablelen** - bieżąca długość tablicy definicji klawiatury. **Minchar** (**maxchar**) zapamiętują odpowiednio minimalną (maksymalną) wartość znaków inicjujących sekwencje. Ostatnie trzy zmienne są używane do czytania pliku z definicją klawiatury.

Nowa wersja funkcji **keycode** ma następującą postać:

```
int keycode()
{
    static BOOLEAN first = TRUE, translate;
    int k;
    if (first)
    {
        first = FALSE;
        if ((translate = compile()) && tablelen <= 0)
            fatal("Błąd kompilacji tablicy definicyjnej");
    }
    if (translate)
    {
        while ((k = lookup(table, tablelen, 0)) == -1)
            PSbeep();
        return(k);
    }
    return(cget());
}
```

Kompilacja tablicy odbywa się tylko raz, przy pierwszym wywołaniu funkcji **keycode**. Jeżeli tablica nie zostanie zdefiniowana - wynikiem funkcji **compile** jest wtedy **FALSE**, więc wynikiem jest wartość funkcji **cget**. W przypadku istnienia takiej tablicy wywoływana jest funkcja **lookup**, realizująca wcześniej opisany algorytm przeszukujący.

Funkcja **compile** wczytuje definicje kodów wirtualnych.

```
static BOOLEAN compile()
/* wczytywanie definicji kodów wirtualnych klawiatury */
{
    char * getenv(), buf[100], *s;
    if ((fin = fopen(filename, "r")) == NULL)
        if ((filename = getenv("KBDEF")) == NULL)
            return(FALSE);
        else if ((fin = fopen(filename, "r")) == NULL)
            kfatal("can't open");
    tablelen = 0;
    maxtablelen = TBLGRANULE;
```

```

table = (KEYSEQ **)xmalloc(TBLGRANULE * sizeof(KEYSEQ *));
while (fgets(buf, sizeof(buf), fin) != NULL)
{
    linenum++;
    buf[strlen(buf) - 1] = '0'; /* usuń nową linię */
    s = strtok_own(buf, "#");
    if (s != NULL && s[0] != '0')
    {
        tablelen++;
        doline(s);
    }
}
if (ferror(fin))
    kfatal("Błąd czytania");
fclose(fin);
qsort((char *)table, tablelen, sizeof(KEYSEQ *), compare);
linenum = -1;
return(TRUE);
}

```

Statyczna zmienna *filename* zawiera ciąg "kbdef.tbl". Jeśli plik o tej nazwie nie znajduje się w bieżącej kartotece, to do odnalezienia nazwy pliku, zawierającego tablicę kodów wirtualnych, używana jest wartość zmiennej systemowej *KBDEF*. Brak pliku o podanej nazwie powoduje zwrócenie przez funkcję *compile* wartości *FALSE*. Kody otrzymywane z funkcji *keycode* są wówczas równe wartościom wywoływanej funkcji *cget*. W przypadku błędu podczas otwierania pliku wywoływana jest funkcja *fatal*, która spełnia rolę informacyjną o zdarzeniach w systemie. Po poprawnym otwarciu następuje alokacja tablicy wskaźników do struktur *KEYSEQ* o długości *TBLGRANULE*. Same struktury są dynamicznie alokowane w miarę potrzeby w funkcji *doline*. Po napotkaniu znaku końca pliku, wywoływana jest standardowa funkcja *qsort* do posortowania tablicy. Funkcji wewnętrznej *compare* przekazuje się dwa wskaźniki wskazujące na elementy sortowanej tablicy, czyli wskaźniki na wskaźniki struktury *KEYSEQ*.

```

static int compare (a, b)
KEYSEQ **a;           /* wskaźnik na pierwszy element */
KEYSEQ **b;           /* wskaźnik na drugi element */
{
    int i;
    for (i = 0; i < (*a)->len; i++)
    {

```

```

    if (i >= (*b)->len)
        return (1);
    if ((*a)->seq[i] != (*b)->seq[i])
        return(((a)->seq[i] & 0377) - ((*b)->seq[i] & 0377));
    }
    return(i >= (*b)->len ? 0 : -1);
}

```

Operator '&' stosowany jest w celu uniknięcia propagacji znaku, która uniemożliwia poprawne wyznaczenie wartości funkcji `compare`.

Funkcja `doline` przetwarza kolejne wiersze z pliku definiującego klawiaturę wirtualną. Zmienia ona w miarę potrzeby rozmiary tablic w sposób opisany poprzednio.

```

static void doline(s)
char *s;
{
    char *tok;
    int tablei, toknum, maxseq, n;
    if (tablelen > maxtablelen)
    {
        maxtablelen += TBLGRANULE;
        if ((table = (KEYSEQ **)realloc((char *)table,
            maxtablelen*sizeof(KEYSEQ *))) == NULL)
            fatal("out of memory");
    }
    tablei = tablelen - 1;          /* wyzeruj indeks */
    maxseq = SEQGRANULE;
    table[tablei] = (KEYSEQ *)xmalloc(sizeof(KEYSEQ));
    toknum = 0;
    while ((tok = strtok(s, " t")) != NULL)
    {
        s = NULL;
        if (tok[0] == '0')          /* pomiń białe znaki */
            continue;
        toknum++;
        if (toknum > maxseq + 1)
        {
            if ((table[tablei] =
                (KEYSEQ *)realloc((char *)table[tablei],
                    sizeof(KEYSEQ) + maxseq)) == NULL)
                fatal("Brak pamięci");
            maxseq += SEQGRANULE;
        }
    }
}

```

```

    }
    if (toknum == 1)
        table[tablei]->code = convert(tok, MAXINT);
    else
    {
        n = convert(tok, 255);
        table[tablei]->seq[toknum - 2] = n;
        if (toknum == 2)
        {
            minchar = MIN(minchar, n);
            maxchar = MAX(maxchar, n);
        }
    }
}

switch (toknum)
{
    case 0:
        kfatal("Pusty wiersz");
    case 1:
        kfatal("Błąd sekwencji");
}

table[tablei]->len = toknum - 1;
}

```

Założyliśmy, że wiersze składają się z sekwencji, czyli tokenów i separatorów. Funkcja `strtok` w nagłówku petli `while` wydziela je z podanego ciągu znaków. Zmienna `toknum` o wartości równej jeden informuje, że przeczytany token jest kodem wirtualnym; dla wartości większej niż jeden, mamy do czynienia z sekwencją wejściową dla danego kodu wirtualnego; natomiast dla wartości równej dwa - sprawdzane są i aktualizowane w miarę potrzeby zmienne `minchar` i `maxchar`. Zmienna `toknum` odgrywa też inną rolę. Jest ona sygnalizatorem realokacji pamięci dla tablicy `seq`, jeśli jest ona niewystarczająca dla całej sekwencji wejściowej.

Każdy token przekształcany jest z formatu znakowego na format binarny. Funkcja ta musi uwzględnić wszystkie dopuszczalne sposoby jej definiowania, a więc obsługiwać znaki specjalne, takie jak: `^`, `+`, `0`. Rozpatrując na początku długość tokena, możemy podjąć odpowiednią akcję. I tak, jeśli długość tokena jest równa 1, to wynikiem jest kod ASCII tego znaku. Gdy wynosi ona 2, możliwe jest wystąpienie znaku kontrolnego `''` na początku tokena, co powoduje późniejszą konwersję drugiego znaku. W pozostałych przypadkach token jest ciągiem cyfr

zapisanych w systemie dziesiętnym lub ósemkowym. Korzystamy wówczas z pomocniczej funkcji `cvint`. Jej wynikiem jest wartość dziesiętna wejściowego ciągu.

Do omówienia pozostała jeszcze tylko funkcja `lookup`. Wywołanie jej w funkcji `keycode` ma następującą postać:

```
while ((k = lookup(table, tablelen, 0)) == -1)
    PSbeep();
```

Argumenty funkcji definiują przeszukiwaną podtablicę: wskaźnik na jej początek, jej długość i liczbę zanalizowanych już znaków sekwencji. Po jej wywołaniu w funkcji `keycode` przeszukiwana jest cała tablica, zatem pierwszy argument jest adresem startowym i wskazuje na początek tablicy `table`. Drugi argument jest równy jej całkowitej długości, a trzeci ma oczywiście wartość 0, ponieważ nie został przeczytany jeszcze ani jeden znak.

Oto definicja funkcji `lookup`:

```
static int lookup(t, tlen, index)
KEYSEQ *t[];           /* adres startowy dla podtablicy */
int tlen;              /* długość podtablicy */
int index;             /* indeks sekwencji wejściowej */
{
    int i, c, seqc, start, end;
    if (tlen <= 0)
        return(-1);
    if (tlen == 1 && index >= t[0]->len)
        return(t[0]->code);
    if ((c = cget()) == K_CTL_BSLSH)
    {
        /* "wyjście bezpieczeństwa" */
        cleanup();
        exit(0);
    }
    if (index == 0 && (c < minchar || c > maxchar))
        return(c);
    start = end = -1;
    for (i = 0; i < tlen; i++)
    {
        if (index >= t[i]->len)
        {
            if (i == tlen - 1)
                kfatal("Błąd porządku");
            kfatal("Sekwencje dla kodów %d i %d są dwuznaczne",
                t[i]->code, t[i+1]->code);
        }
    }
}
```

```
    }
    seqc = t[i]->seq[index] & 0377;
    if (seqc == c)
    {
        if (start == -1)
            start = i;
    }
    else
        if (seqc > c)
        {
            end = i - 1;
            break;
        }
    }
    if (end == -1)
        end = tlen - 1;
    if (start == -1)
        if (index == 0)
            return(c);
        else
            return(-1);
    return(lookup(&t[start], end - start + 1, index + 1));
}
```

Opis funkcji `lookup` jest dość złożony i dlatego funkcja ta zasługuje na dokładniejsze omówienie. Rozpoczyna się ona sprawdzeniem poprawności wywołania funkcji. Jeśli długość podtablicy jest ujemna (nie istnieje taka podtablica), to otrzymujemy `-1`. W rzeczywistości taka wartość parametru nie powinna nigdy zostać przekazana, ale doświadczeni programiści wiedzą, że warto jest stosować takie dodatkowe zabezpieczenia. Jeśli długość podtablicy wynosi 1 i cała sekwencja wejściowa została przeanalizowana, to oznacza, że znaleziono już właściwy kod wirtualny. Jest on wartością tej funkcji. W przeciwnym przypadku wczytywany jest z klawiatury znak. Znak `Ctrl-\` jest tzw. wyjściem bezpieczeństwa i używamy go w przypadku niekompletnej czy błędnie określonej definicji klawiatury. Wyjście to może się okazać przydatne np. w sytuacji, gdy program usiłuje wczytać niezdefiniowany kod wirtualny. Jeśli wczytany znak jest pierwszy z sekwencji (indeks równa się zeru) i nie znajduje się w przedziale określonym przez `minchar` i `maxchar`, to jest on zwracany jako wartość funkcji.

Jeżeli długość wczytanego ciągu osiągnęła już długość sekwencji a tablica zawiera jeszcze kilka opisów kodów wirtualnych, to oznacza to ich błędne zdefiniowanie i powoduje wyprowadzenie stosownego komunikatu oraz zakończenie pracy programu. Z sytuacją taką mamy do czynienia, np. po wpisaniu opisów kodów wirtualnych w postaci:

301 a c b

305 a c b a

Po pobraniu z klawiatury znaków a c b niemożliwe jest rozstrzygnięcie, czy należy zwrócić kod wirtualny 301 czy też wczytać następny znak i ewentualnie zwrócić kod 305.

Zmienna **start** i **end** są wykorzystane do zapamiętania indeksu odpowiednio pierwszego i ostatniego opisu kodu wirtualnego, mającego sekwencję zgodną z wczytanymi znakami. Na początku zmienne te otrzymują wartość -1. Jeśli zmienna **end** po przejrzeniu podtablicy nie zmieniła swojej wartości, to podstawiamy pod nią ostatni indeks w przeszukiwanej podtablicy. Jeśli zmienna **start** nie zmieni swojej wartości, oznacza to, że żadna z sekwencji kodów podtablicy nie jest zgodna z wczytanym znakiem. Jest to sytuacja poprawna tylko wtedy, gdy wczytany znak jest pierwszy w sekwencji (indeks==0), w przeciwnym przypadku wynikiem funkcji jest -1, sygnalizując w ten sposób wystąpienie błędu.

Na koniec wywoływana jest rekurencyjnie funkcja **lookup** do przeszukania następnej podtablicy. Pozwala to na poszukiwanie sekwencji o dowolnej długości. Ten typ rekurencji można łatwo zastąpić iteracją, co zwiększyłoby nieco szybkość działania programu. W przypadku przetwarzania znaków wprowadzonych z klawiatury jest to jednak bez znaczenia i dlatego wykorzystano tu rekurencję - rozwiązanie uznawane zwykle za bardziej eleganckie.

5.4. Podsumowanie

W rozdziale tym, na samym początku oprogramowano funkcje niskiego poziomu, by uzyskać dostęp do klawiatury niezależnie od systemu operacyjnego. Naciśnięcie tego samego klawisza na różnych terminalach powodować może wysłanie do programu różnych kodów. Zdefiniowanie tzw. kodów wirtualnych pozwala na uniezależnienie programu od rodzaju stosowanej klawiatury. Funkcje kolejnej warstwy (**Kready** i **Kget**) przekształcają kody rzeczywiste w kody wirtualne. Przedstawiono dwa sposoby implementowania tych funkcji:

- statyczny, wykorzystujący rozbudowaną instrukcję **switch** lub tablicę w programie źródłowym

- dynamiczny przez zewnętrzną tablicę definicji kodów wirtualnych; pozwala to na zmianę sposobu obsługi klawiatury bez potrzeby ingerencji w kod źródłowy.

Trzecią warstwę obsługi klawiatury tworzą funkcje użytkowe. Dokładne ich omówienie wykracza poza ramy publikacji i dlatego rozdział ten zakończony zostanie podaniem kilku propozycji funkcji tej warstwy. Są nimi np. funkcje:

1. Tworzące makra, tj. pozwalające użytkownikowi na zastąpienie sekwencji wciśniętych klawiszy przez jeden kod.
2. Obsługujące tzw. hot keys (gorące klawisze); pełnią one rolę podobną jak makra, ale mogą być definiowane tylko przez programistę a nie przez użytkownika systemu.
3. Symulujące wciśnięcie sekwencji klawiszy; funkcja ta wpisuje argument na stos; funkcja `Kget` pobiera wówczas kody z klawiatury, jeżeli stos ten jest pusty.
4. Obsługujące filtr ignorujący wciśnięcie wybranych klawiszy; funkcja ta pozwala np. na prostą kontrolę poprawności wprowadzanych danych.
5. Zmieniające standard obsługi klawiatury; użytkownik systemu mającego kilka definicji kodów wirtualnych może sam zdecydować np., czy pragnie obsługiwać edytor w standardzie WordPerfect czy też Wordstar.

Funkcje tworzące dwie pierwsze warstwy programowania pozwalają na szybkie tworzenie funkcji użytkowych.

Rozdział 6

PRZETWARZANIE DANYCH TEKSTOWYCH

Wzrost pojemności pamięci pomocniczych, szybkości pracy komputerów i oczekiwań użytkowników sprawił, że obecnie coraz częściej systemy informacyjne przetwarzają nie tylko sformatowane opisy obiektów, ale także ich opisy słowne. Obecnie dostępnych jest kilka pakietów ułatwiających implementację w języku C któregoś z modeli sformatowanych danych (najczęściej jest to model relacyjny), brak jest jednak podobnych narzędzi służących do przetwarzania tekstów.

Kolejne podrozdziały omawiają podstawowe problemy, z jakimi mamy do czynienia podczas przetwarzania tekstów. Kolejno zostanie omówione przechowywanie, przeszukiwanie i opisywanie tekstów w języku naturalnym.

6.1. Przechowywanie danych tekstowych

Jedną z głównych cech różniących opis słowny od opisu sformatowanego jest to, że jego długość może zmieniać się w bardzo dużych granicach i ponadto zmieniać w trakcie pracy systemu. Zapamiętanie tekstów w rekordach stałej długości jest zatem niemożliwe. Do przetwarzania danych zmiennej długości doskonale nadają się funkcje `alloc`, `free`, `realloc`, lecz operują one jedynie na danych zapisanych w pamięci operacyjnej. Podczas zapisu danych do pamięci zewnętrznej można naturalnie przerzucić odpowiedzialność za obsługę zmiennej długości tekstu na system operacyjny – wystarczy zapamiętać każdy z nich w odrębnym pliku. Rozwiązanie takie jest jednak bardzo nieefektywne. Prowadzi ono do tworzenia wielkiej liczby plików, niewykorzystania znacznej części zajętej pamięci i spowolnienia pracy systemu.

Przedstawione poniżej rozwiązanie polega na zapamiętaniu takich danych w bazie mającej rekordy zmiennej długości. Struktura jej będzie omówiona dokładniej w następnym podrozdziale. Funkcje obsługujące tę bazę pozwalają na wykonanie wszystkich podstawowych operacji tworząc warstwę DZD – obsługi danych zmiennej długości. Warstwa ta może być

naturalnie wykorzystana i do innych celów, w dalszej części rozdziału pokazane jest, jak można ją użyć do tworzenia tablicy inwersyjnej przyspieszającej przeszukiwanie tekstów (patrz p. 6.3.2).

6.1.1. Omówienie funkcji

Warstwa DZD umożliwia zapamiętanie w jednej bazie wielu rekordów. Długość każdego z nich może się zmieniać (jedyne ograniczenia nakłada tu wielkość dostępnej pamięci operacyjnej i zewnętrznej), nie zmienia się natomiast przypisany mu unikatowy identyfikator. W razie skasowania rekordu gwarantowane jest, że jego identyfikator nie zostanie przypisany żadnemu innemu, nowo wprowadzanemu rekordowi. Identyfikatorami są liczby typu long. Funkcje te nie ingerują w wewnętrzną strukturę rekordu.

Wszystkie funkcje tej warstwy są typu int lub long. W razie wystąpienia błędu zwracają one wartość **ERR**. Definicja tej, jak i innych stałych, deklaracje funkcji wraz z definicjami wykorzystywanych struktur, są umieszczone w pliku nagłówkowym `dzd.h`. Warstwa DZD zawiera łącznie 13 funkcji.

Pierwsza ich grupa służy do wykonywania operacji na całej bazie. Są nimi: `dzd_ini`, `dzd_open` i `dzd_close`.

```
int dzd_ini(w, dl_rek, nazwa)
/* zakładanie bazy dzd */
ops_dzd *w;
/* wskaźnik na strukturę opisującą bazę */
unsigned dl_rek;
/* długość "fizycznego" rekordu w bazie */
char *nazwa
/* nazwa bazy */
```

Funkcja ta służy do zakładania bazy. Jeżeli istnieje już inna baza o podanej nazwie, to jest ona skasowana przed utworzeniem nowej. Jednocześnie możemy wykorzystywać wiele baz, struktura `*w` zawiera wszystkie dane niezbędne do wykonania każdej funkcji z tej warstwy. Zalecane jest uwzględnienie, podczas określania wartości parametru `dl_bloku`, średnich rozmiarów rekordów. Rekordy w bazie są zapisywane jako listy bloków stałej długości równej `dl_rek`. Zwiększenie jego wartości przyspiesza działanie funkcji, lecz powoduje jednocześnie zwiększenie rozmiarów plików. Nazwa bazy jest to pozbawiona rozszerzenia nazwa pliku, dopuszczalna w stosowanym systemie operacyjnym. Funkcja `dzd_ini` zakłada dwa pliki, o nazwach różniących się tylko rozszerzeniami, z których jeden zawiera dane, a drugi przyspiesza ich udostępnianie (patrz p. 6.1.2).

```

int dzd_open (w, nazwa)
/* otwieranie istniejącej bazy dzd */
ops_dzd *w;
/* wskaźnik na strukturę opisującą bazę */
char *nazwa
/* nazwa otwieranej bazy */

```

Funkcja **dzd_open** otwiera bazę założoną przez **dzd_ini**. Dane robocze umieszczone w plikach tworzących bazę są wykorzystane do zapelnienia struktury ***w**. Nazwa bazy nie może zawierać rozszerzenia. Po otwarciu bazy identyfikuje się ją jedynie przez strukturę ***w**.

```

int dzd_close (w)
/* zamykanie bazy dzd */
ops_dzd *w;
/* wskaźnik na strukturę opisującą bazę */

```

Funkcja **dzd_close** zamyka bazę, dane robocze są przepisywane ze struktury ***w** do plików.

Następna grupa wykonuje działania na pojedynczym rekordzie bazy. Działania te są wykonywane zawsze na bieżącym rekordzie bazy.

```

long dzd_ins (w, dl, tresc)
/* dopisanie do bazy nowego rekordu */
ops_dzd *w;
/* wskaźnik na strukturę opisującą bazę */
unsigned dl;
/* długość w bajtach dopisywanego rekordu */
char *tresc;
/* wskaźnik na zawartość dopisywanego rekordu */

```

Funkcja **dzd_ins** dopisuje nowy rekord do bazy. Staje się on rekordem bieżącym. Wartością tej funkcji jest unikatowy identyfikator przypisany przez funkcje warstwy dzd temu rekordowi.

```

int dzd_get (w, gdzie)
/* wczytanie rekordu z bazy */
ops_dzd *w;
/* wskaźnik na strukturę opisującą bazę */
dzd *gdzie;
/* wskaźnik na strukturę opisującą rekord */

```

Funkcja **dzd_get** wpisuje aktualny rekord do struktury ***gdzie**. Gdy funkcja ta jest poprawnie wykonana wówczas funkcja ta alokuje obszar pamięci równy długości rekordu, wpisuje tam jego treść, a następnie wskaźnik na ten obszar umieszcza w strukturze ***gdzie**. Wartością zwracaną przez funkcję jest wtedy OK. Błędne dane lub brak możliwości zaalokowania pamięci powodują zwrócenie wartości **ERR**. Użytkownik musi sam zadbać o zwolnienie pamięci przydzielonej przez funkcję **dzd_get**.

```

int dzd_rep (w, dl, tresc)
/* zmiana treści bieżącego rekordu */
ops_dzd *w;
/* wskaźnik na strukturę opisującą bazę */
unsigned dl;
/* długość zmienianych danych */
char *tresc;
/* wskaźnik na nową zawartość rekordu */

```

Funkcja **dzd_rep** kasuje starą zawartość bieżącego rekordu, wpisując na jej miejsce nową, określoną przez parametry **dl** i **treść**.

```
int dzd_set (w, ident) /* ustawienie wskaźnika bieżącego rekordu */
ops_dzd *w;           /* wskaźnik na strukturę opisującą bazę */
long ident;           /* identyfikator rekordu */
```

Bieżącym rekordem staje się rekord o identyfikatorze równym wartości **ident**. Jeżeli rekord taki istnieje, to wartością **dzd_set** jest **OK**, w przeciwnym przypadku **ERR**. Funkcja ta udostępnia nam dowolny rekord bazy pod warunkiem, że znamy jego identyfikator. Często jednak konieczne jest przeglądnięcie całej zawartości bazy. Umożliwiają to funkcje **dzd_fst** i **dzd_nxt**:

```
int dzd_fst (ops_dzd *w)
/* ustawianie wskaźnika na pierwszy rekord */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */
```

Bieżącym rekordem staje się pierwszy (to jest mający najmniejszy identyfikator), nieskasowany rekord bazy. Jeżeli rekord taki istnieje, to zwracane jest **OK**, w przeciwnym przypadku (baza jest pusta) zwracane jest **ERR**.

```
int dzd_nxt (w)
/* przesunięcie wskaźnika na następny rekord */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */
```

Bieżącym rekordem staje się następny, nieskasowany rekord bazy. Jeżeli rekord taki istnieje, to zwracane jest **OK**, w przeciwnym przypadku (baza nie zawiera już więcej rekordów) zwracane jest **ERR**.

```
int dzd_del (w)
/* kasowanie bieżącego rekordu */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */
```

Aktualny rekord jest usuwany z bazy. Wystąpienie błędu funkcja ta sygnalizuje zwracając **ERR**, w przeciwnym przypadku jest to **OK**.

Ostatnia grupa funkcji służy do uzyskania dokładniejszych informacji o błędzie. Samo stwierdzenie faktu jego wystąpienia jest zwykle niewystarczające. Kod ostatniego błędu uzyskamy wywołując funkcję **dzd_err**.

```
int dzd_err () /* podaj kod ostatniego błędu */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */
```

Kody błędów są zdefiniowane w pliku nagłówkowym **zdz.h**. Zamiast kodu błędu można uzyskać opis przyczyny jego wystąpienia:

```

char *dzd_err_txt(ops_dzd *w);
                /* podaj opis błędu */
ops_dzd *w;      /* wskaźnik na strukturę opisującą bazę */

    Oprogramowanie warstwy "zapomni" o wystąpieniu błędu dopiero po
    jego skasowaniu. Służy do tego celu funkcja dzd_err:
int dzd_err_clr(ops_dzd *w);
                /* kasuj błąd */
ops_dzd *w;      /* wskaźnik na strukturę opisującą bazę */

```

6.1.2. Prezentacja kodu funkcji

Definicja struktury **ops_dzd** opisującej stan bazy jest umieszczona w pliku nagłówkowym **dzd.h**:

```

typedef struct
{
int ost_bl;      /* kod ostatniego błędu */
char **kody_bl;  /* tablica wskaźników na opisy błędów */
long nast_ident; /* pierwszy wolny identyfikator */
long akt_ident;  /* identyfikator bieżącego rekordu */
long dl_pliku;   /* długość pliku z rekordami */
long nast_wolny; /* pierwszy wolny "fizyczny" blok */
unsigned dl_bloku; /* długość fizycznego bloku zapisu rek. */
FILE *idx_zb;    /* wskaźnik na plik z identyfikatorami */
FILE *txt_zb;    /* wskaźnik na plik z rekordami */
char *nazwa;     /* nazwa bazy */
} ops_dzd;

```

Każda z wykorzystywanych baz ma własną strukturę typu **ops_dzd**, wszystkie one korzystają z tej samej tablicy z opisami błędów. Do opisu rekordów służy struktura **dzd**:

```

typedef struct
{
long ident;      /* identyfikator rekordu */
unsigned dl_txt; /* długość rekordu w bajtach */
char *txt;       /* wskaźnik na obszar z treścią rekordu */
} dzd;

```

Plik **dzd.h** zawiera także deklaracje funkcji zewnętrznych tej warstwy. Kod źródłowy funkcji realizujących warstwę **dzd** jest zapisany w pliku **dzd.c**.

Dane w bazie są zapisane w dwu plikach. Nazwy ich są utworzone przez dodanie do nazwy bazy rozszerzeń:

```
static char idx_ext[]=".dzi"; /* dla plików z treścią rekordów */
static char txt_ext[]=".dzd"; /* dla plików indeksowych */
```

W pliku o nazwie z rozszerzeniem txt_ext są umieszczone rekordy logiczne. Każdy z nich jest zapisywany jako lista rekordów fizycznych. Rekordy fizyczne mają stałą długość, określoną przez użytkownika w momencie zakładania bazy. Rekord fizyczny składa się z dwóch części: adresu następnego rekordu fizycznego oraz jego treści. Adres o wartości = -1 oznacza, że dany rekord jest już na końcu listy. Na początku rekordu logicznego są zapisane znakowo: identyfikator rekordu oraz jego długość. Pozwala to na kontrolę poprawności wykonywania funkcji i odtworzenia (przynajmniej częściowego) zawartości bazy w wypadku uszkodzenia plików bazy.

Drugi plik pełni rolę zewnętrznej tablicy. Jest ona indeksowana identyfikatorami rekordów logicznych, a wartości jej są równe początkowym adresom odpowiednich rekordów w pliku z treścią. Wartość -1 wskazuje, że rekord o tym identyfikatorze został już skasowany. Plik ten zawiera nagłówek, w którym przechowywane są podstawowe dane o bazie, takie jak: długość bloku fizycznego, następny identyfikator do wykorzystania i adres pierwszego wolnego bloku fizycznego do zapisu. Długość tego nagłówka określa stała **START_IDX** zaś adres logicznego rekordu o identyfikatorze x jest oddalony o $IDX(x)$ od początku tego pliku:

```
#define START_IDX 20
#define IDX(x) (START_IDX + sizeof(long)*(x))
#define BLAD(n) { w->ost_bl=n; return(ERR); }
#define DL_MIN 20
#define MAX_L_BLOKOW 100
```

Makro **BLAD** stosowane jest po stwierdzeniu błędu w wykonaniu funkcji, zaś stała **DL_MIN** wyznacza minimalną długość bloku fizycznego. Musi być on wystarczająco duży, by pomieścić dane kontrolne zapisywane w pierwszym rekordzie listy. Ostatnia ze stałych **MAX_L_BLOKOW** wyznacza maksymalną liczbę bloków fizycznych tworzących jeden rekord. Jest ona stosowana do sprawdzania poprawności zapisu i odczytu danych. Tablica **dzd_kody_bl** zawiera słowne opisy przyczyn błędów:

```
static char *dzd_kody_bl[]=
{
    "OK",
    "Brak_ident",
    "Blad_w_pliku",
    "Brak_pliku",
```

```

    "Ident_skasowany",
    "Brak_pao",
};

```

Oprócz tego umieszczono tam jeszcze zmienna `koniec_bl`:

```
static long koniec_bl=-1L;
```

Wartość oznaczająca koniec listy rekordów fizycznych lub skasowanie identyfikatora została wpisana do zmiennej, a nie została zdefiniowana – jak to zwykle jest czynione – jako stała. Powód jest prosty: jest ona często wpisywana do pliku za pomocą funkcji `fwrite`, która wymaga podania wskaźnika na obszar z zapisywaną wartością. Poniżej omówiony zostanie kod trzech typowych funkcji tej warstwy: `dzd_ini`, `wolny_blok` i `dzd_get`.

```

int dzd_ini (ops_dzd *w, unsigned dl_bloku, char *nazwa)
{
    char texty[MAXPATH];
    char indxy[MAXPATH];
    strcpy(texty,nazwa);
    strcat(texty,txt_ext);
    strcpy(indxy,nazwa);
    strcat(indxy,idx_ext);
    if ( dl_bloku<=DL_MIN || (w->txt_zb=fopen(texty,"wb+"))==NULL)
        BLAD(3)
    if ( (w->idx_zb=fopen(indxy,"wb+"))==NULL)
        BLAD(3)
    w->nast_wolny=koniec_bl;
    w->nast_ident=0L;
    fwrite(&dl_bloku,(size_t) sizeof(int),(size_t)1,w->idx_zb);
    fwrite(&w->nast_ident,(size_t)sizeof(long),(size_t)1,w->idx_zb);
    fwrite(&w->nast_wolny,(size_t)sizeof(long),(size_t)1,w->idx_zb);
    if (ferror(w->idx_zb)!=0)
        BLAD(2)
    w->dl_pliku=0L;
    w->dl_bloku=dl_bloku;
    w->akt_ident=-1L;
    return(OK);
}

```

Funkcja `dzd_ins` tworzy nazwy obu plików bazy przez dopisanie odpowiednich rozszerzeń do nazwy bazy i następnie otwiera oba pliki w trybie "wb+". Stosowany jest tu tryb binarny, bowiem konwersja znaków dokonywana przez tryb tekstowy powoduje błędy we wczytywaniu i zapisie danych. Po sprawdzeniu poprawności parametru `dl_bloku` następuje

wpisanie początkowych wartości do struktury **w* oraz do nagłówka pliku indeksowego. Poprawność zapisu danych do pliku jest kontrolowana za pomocą funkcji *ferror*.

Wolny_blok jest funkcją lokalną tej warstwy. Zadaniem jej jest dostarczanie adresu pierwszego wolnego bloku fizycznego. Wszystkie niewykorzystane rekordy fizyczne w bazie tworzą jedną listę. Usunięcie rekordu z bazy powoduje dopisanie do niej jego rekordów, podczas zapisu danych zaś brane są przede wszystkim rekordy fizyczne z tej listy. Jeśli jest ona pusta, to są one dopisywane na koniec pliku.

```
static long wolny_blok(ops_dzd *w)
{
    long wyn;
    if (w->nast_wolny != !oniec_bl)
    {
        wyn = w->dl_pliku;
        w->dl_pliku += w->dl_bloku + sizeof(long);
    }
    else
    {
        wyn = w->nast_wolny;
        fseek(w->txt_zb, wyn, SEEK_SET);
        fread(&w->nast_wolny, (size_t)sizeof(long), (size_t)1,
              w->txt_zb);
    }
    return(wyn);
}
```

Przyjęty sposób działania funkcji *wolny_blok* sprawia, że początkowo rekordy bazy są zapisywane w następujących po sobie blokach fizycznych. Z biegiem czasu jednak, w miarę jak coraz więcej rekordów jest modyfikowanych lub kasowanych, następuje fragmentaryzacja ich zapisu. Oznacza to, że rekord logiczny jest zapisany w wielu, oddalonych od siebie blokach fizycznych. Spowoduje to zmniejszenie szybkości zapisu i odczytu danych. Celowa jest wtedy reorganizacja bazy, co osiągnąć można przepisując wszystkie rekordy logiczne do nowej bazy.

Ostatnią z prezentowanych funkcji jest *dzd_get*:

```
int dzd_get (ops_dzd *w, dzd *gdzie)
{
    char nagl[DL_MIN], *r;
    long ident, ak_id, start_adr, next_adr;
    unsigned dl;
```



```

ident=w->akt_ident;
if (ident<0 || ident>w->nast_ident)
    BLAD(1);
fseek(w->idx_zb,IDX(ident),SEEK_SET);
fread(&start_adr,(size_t)sizeof(long), (size_t) 1, w->idx_zb);
if (start_adr<0)
    BLAD(1);
fseek(w->txt_zb,start_adr,SEEK_SET);
fread(&next_adr,(size_t)sizeof(long), (size_t) 1, w->txt_zb);
fread(nagl,DL_MIN,1,w->txt_zb);
sscanf(nagl," %ld %u",&ak_id,&dl);
if (ak_id!=ident)
    BLAD(1);
gdzie->ident=ident;
gdzie->dl_txt=dl;
if ((gdzie->txt_r=(char *)malloc(dl))==NULL)
    BLAD(4);
fread(r,w->dl_bloku-DL_MIN, (size_t) 1, w->txt_zb);
r+=w->dl_bloku-DL_MIN;
bezpieka=MAX_L_BLOKOW;
while (next_adr!=koniec_bl && --bezpieka>0)
{
    fseek(w->txt_zb,next_adr,SEEK_SET);
    fread(&next_adr,(size_t)sizeof(long), (size_t) 1, w->txt_zb);
    fread(r,w->dl_bloku, (size_t) 1, w->txt_zb);
    r+=w->dl_bloku;
}
if (ferror(w->idx_zb)!=0 || ferror(w->txt_zb)!=0)
    BLAD(2)
if (bezpieka>0)
    return(OK);
else
    BLAD(2);
}

```

Funkcja **dzd_get** sprawdza, czy poszukiwany rekord jest umieszczony w bazie, i jeżeli tak jest, to wczytuje pierwszy jego blok. Zadaniem **sscanf** jest konwersja początkowych bajtów jego treści z zapisu znakowego na binarny. Uzyskane wartości są wykorzystane do kontroli poprawności wczytanych danych oraz do określenia długości rekordu. Następnie alokowany jest obszar pamięci na treść rekordu i dopisywane

do niego są kolejne, tworzące go bloki fizyczne. Jeżeli rozmiary danych przekraczają zaalokowaną wielkość obszaru pamięci, to oznacza to wystąpienie błędu w danych.

6.1.3. Przykład

Przedstawiony w tym podrozdziale program wczytuje kolejne zdania z wejściowego pliku, zapisując je w bazie. Zawartość bazy jest następnie wypisana na ekranie:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <alloc.h>
#include <dir.h>
#include <dzd.h>

void wpisz_dane(FILE *in, ops_dzd *w)
{
    char zdanie[500], koniec[2];
    koniec[1]=0;
    while (fscanf(in, " %500[^\n?;%[.!?]]", zdanie, &koniec[0])!=2)
    {
        strcat(zdanie, koniec);
        dzd_ins(w, strlen(zdanie), zdanie);
    }
}

void wypisz_dane(ops_dzd *w)
{
    dzd dane;
    dzd_fst(w);
    do
    {
        dzd_get(w, &dane);
        puts(dane.txt);
    } while(dzd_nxt(w)!=OK);
}

void main(int lp, char *par[])
{
    FILE *in;
    ops_dzd w;
    if (lp!=3)
    {
```

```

    puts("zaklada dane_wejscowe nazwa_bazy");
    exit(1);
}
if ((in=fopen(par[1],"r"))==NULL)
{
    puts("brak danych wejsciowych");
    exit(2);
}
if (dzd_ini(&w,30,par[2])==ERR)
{
    puts("blad w zakladaniu bazy tekstow");
    exit(3);
}
wpisz_dane(in,&w);
fclose(in);
wypisz_dane(&w);
dzd_close (&w);
}

```

Jak widać, zarówno nazwa pliku z danymi wejściowymi, jak i nazwa bazy, są pobierane z linii zlecenia. Na uwagę zasługuje sposób, w jaki dane wejściowe są dzielone na zdania. Wykorzystano w tym celu funkcję *fscanf*. Ciąg formatujący ma postać: " %500[^\?%[^\?]" . Spacja na jego początku sprawia, że dowolna liczba znaków niedrukowalnych (tzw. white space), występujących między zdaniami, jest ignorowana. Umieszczenie z kolei sekwencji znaków %[x...y] oznacza polecenie wczytania ciągu składającego się wyłącznie ze znaków x...y, a %[^\x...y] ciągu nie zawierającego ani jednego z wymienionych znaków.

Użyteczność funkcji *fscanf* ogranicza nieco fakt, że zmienna nie może określać długości wczytywanego ciągu, tak samo jak w przypadku *fprintf*. W przypadku stosowania kompilatora Borland dodatkowym utrudnieniem jest niezastępowanie przez preprocesor makr występujących wewnątrz ciągów znaków. Powoduje to, że nie można użyć makra do zdefiniowania ograniczników zdania. Podczas pisania funkcji wczytujących dane o różnej długości można przygotować właściwą postać ciągu formatującego, np. za pomocą funkcji *sprintf*.

6.2. Opis tekstu

Systemy wykorzystujące dane sformatowane dysponują z reguły metodami dokładnej kontroli poprawności wprowadzanych danych, sprawdzane jest tam np. czy dany ciąg znaków jest dopuszczalną datą urodzenia lub kodem miejsca pracy. Ułatwia to znacznie wyszukiwanie

danych. W przypadku systemów z danymi niesformatowanymi jest to o wiele trudniejsze. Wiele z nich zawiera teksty w języku naturalnym, a ten nie poddaje się łatwo normalizacji i opisowi.

W programach narzędziowych, np. edytorach czy debuggerach, często do opisu poszukiwanego tekstu są stosowane wyrażenia regularne (patrz p. 7.4). Wyrażenia te nie nadają się jednak do opisu tekstów w języku naturalnym. Są one wystarczające do identyfikowania stosunkowo prostych języków formalnych, zwanych regularnymi. W ich ramach można opisać np. nazwy zmiennych, lecz język naturalny ma zupełnie inną, bardziej skomplikowaną strukturę.

Prowadzone od dziesiątków lat prace nad językiem naturalnym wykazały, jak bardzo skomplikowane jest przeprowadzenie syntaktycznej i semantycznej analizy, fragmentu języka naturalnego mającego jakiegokolwiek znaczenie praktyczne. Jest to zadanie daleko wykraczające poza zakres omawianego tu materiału. Dlatego też zaproponowany język opisu wykorzystuje jedynie informacje o występowaniu ciągów znaków, bez przeprowadzania dokładniejszej analizy tekstu.

6.2.1. Syntaktyka i semantyka języka opisu

Najprostszymi wyrażeniami języka opisu tekstu są wyrażenia elementarne. Mogą one przyjąć jedną z dwóch postaci:

`ciąg_znaków` lub `dokl(ciąg_znaków)`.

Ciąg znaków jest to dowolna sekwencja znaków ograniczona znakami ". Jeżeli pragniemy umieścić znak " w środku ciągu_znaków, to musi być on zapisany podwójnie. W celu uproszczenia zapisu przyjmiemy, że ciągi_znaków zaczynające się od litery, nie zawierające znaków spacji i różne od sekwencji `dokl`, mogą być zapisane bez potrzeby ograniczania ich przez znaki ", tzn. zapisy "znaki" i znaki są sobie równoważne. Oba te wyrażenia elementarne różnią się znaczeniem. Pierwsze z nich oznacza przybliżoną, a drugie dokładną zgodność znaków w tekście i wzorcu.

Każde wyrażenie elementarne jest wyrażeniem języka. Ponadto, jeżeli w_1 i w_2 są wyrażeniami, to są nimi także:

$(^w_1)$, $(w_1 * w_2)$ i $(w_1 + w_2)$.

Zakładamy że operator $^$ ma priorytet najwyższy, a $+$ najniższy. Pozwala to na pomijanie nadmiarowych nawiasów w zapisach wyrażeń, tak samo jak dla wyrażeń arytmetycznych.

Niech T oznacza zbiór tekstów, a w dowolne wyrażenie. Przez $S(T, w)$ oznaczać będziemy wartość wyrażenia w dla tekstów T . Sposób obliczania wartości funkcji $S(T, w)$ jest zdefiniowany rekurencyjnie. Wartością $S(T, w)$, dla w będącego wyrażeniem elementarnym w postaci $\text{dokl}(\text{ciąg_znaków})$ jest podzbiór T , złożony ze wszystkich tekstów zawierających sekwencję znaków dokładnie równą ciąg_znaków . Dla drugiej postaci wyrażenia elementarnego nie jest wymagana pełna zgodność, do wybrania tekstu wystarczy, by znalazł się w nim ciąg znaków o podobieństwie przekraczającym wartość graniczną. Przykładowy sposób wyznaczania podobieństwa wyrazów opisano w p. 6.3.3.

Wartości pozostałych wyrażeń są obliczane następująco:

$$S(T, (^w)) = T - S(w)$$

$$S(T, (w_1 + w_2)) = S(T, w_1) \text{ suma } S(T, w_2)$$

$$S(T, (w_1 * w_2)) = S(T, w_1) \text{ iloczyn } S(T, w_2)$$

Przykłady:

1. $S(T, (\text{komputer} * \text{dokl}(\text{"język naturalny"})))$ = wszystkie teksty T zawierające sekwencję znaków wystarczająco podobną do słowa komputer oraz sekwencję dokładnie równą ciągowi "język naturalny".
2. $S(T, (\text{obrazy} * (\text{binarne} + \text{kolorowe})))$ = wszystkie teksty T zawierające sekwencję znaków wystarczająco podobną do słów obrazy i binarne lub kolorowe.

6.2.2. Obliczanie wartości wyrażeń języka opisu tekstów

Obliczanie wartości wyrażeń opisujących teksty jest podobne do obliczania wartości wyrażeń arytmetycznych. Dlatego też wyrażenia te są przed przystąpieniem do obliczania ich wartości zamieniane z postaci normalnej (infiksowej) na ONP - odwrotną notację polską.

Jeżeli obliczanie wartości dla operatorów $\wedge$, $*$ i $+$ nie jest trudne - wymaga to bowiem wykonania zwykłych operacji teoriomnogościowych, to obliczenie wartości wyrażeń elementarnych może być bardzo czasochłonne. Sposoby przyspieszania tego procesu omówiono w p. 6.3.

Zamieszczone poniżej funkcje do zamiany infiksowego zapisu wyrażeń języka opisu tekstów na zapis w ONP mogą być w prosty sposób zmodyfikowane tak, by analizować dowolny inny infiksowy język, zwłaszcza język wyrażeń arytmetycznych. Wyznaczenie wartości przekształconego na ONP wyrażenia, jako powszechnie znane i bardzo proste, zostało pominięte.

6.2.2.1. Zamiana wyrażenia nawiasowego na Odwrotną Notację Polską

Wejściowy ciąg znaków jest dzielony na najmniejsze składniki mające znaczenie, zwane dalej tokenami. Do zamiany wyrażen z postaci infiksowej na ONP służy funkcja `zamień_na_onp`:

```
int zamień_na_onp(txt_inf, op, tab, dl_tab)
char *txt_inf;    /* wskaźnik na wyrażenie w zapisie infiksowym */
operator *op;     /* tablica z opisem stosowanych operatorów */
token *tab;       /* tablica na zapisanie wyrażenia w ONP */
int dl_tab;       /* liczba elementów w tablicy tab */
```

Funkcja ma wartość OK, jeśli zamiana została wykonana, i ERR po wykryciu błędu. Wykrywane są jedynie błędy w zapisie pojedynczych składników wyrażenia, tj. operandów i operatorów. Błędy związane z niewłaściwym użyciem operatorów nie są wykrywane. Opisy operatorów stosowane w wyrażeniach są podane w tablicy `op`. Zmiana wartości tego parametru pozwala na wykorzystanie tej samej funkcji do analizy różnego typu wyrażen. Wynik zamiany jest wpisywany do tablicy `tab`. Jeśli tablica `tab` jest za mała na pomieszczenie zapisu wyrażenia w ONP, to analiza jest przerywana i funkcja zwraca ERR. Na początku pracy funkcja `zamień_na_ONP` wypełnia 0 całą tablicę `tab`. Pozwala to na określenie, ile jej elementów zostało zapisanych.

Oprócz funkcji `zamień_na_onp` ten plik programowy zawiera jeszcze dwie inne funkcje zewnętrzne: `na_stos` i `ze_stosu`. Są one wykorzystywane do obliczania wartości wyrażenia.

```
int na_stos (stos, max, typ, token, wolny)
token *stos;    /* tablica tokenów z zapisem w ONP */
int max;        /* maksymalna liczba tokenów w tablicy stos */
int typ;        /* typ zapisywanego tokena */
char *token;    /* ciąg znaków reprezentujący zapisywany token */
int *wolny;     /* wskaźnik na pierwszy wolny element tablicy stos */
```

Rozpoznawane typy elementów są zdefiniowane w pliku nagłówkowym `onp.h`. Jeżeli rozmiar tablicy `stos` na to pozwala, to dopisywany jest na koniec kolejny element i zwracana jest wartość OK. Jeżeli w tablicy jest brak miejsca na kolejny element, to wartością jej jest ERR.

```
int ze_stosu (token *stos, int *typ, char *tok, int *wolny)
token *stos;    /* tablica z zapisem w ONP */
int *typ;       /* typ pobieranego tokena */
char *token;    /* ciąg znaków reprezentujący pobierany token */
int *wolny;     /* wskaźnik na pierwszy wolny element tablicy stos */
```

Funkcja `ze_stosu` ma wartość `ERR`, jeśli stos nie zawiera żadnego elementu, i `OK` w przeciwnym przypadku.

6.2.2.2. Analiza kodu funkcji

Definicje stałych i typów oraz deklaracje funkcji stosowanych do zamiany wyrażeń z postaci infiksowej na ONP są umieszczone w pliku `onp.h`.

Badane tokeny są opisywane przez strukturę `token`:

```
typedef struct
{
    char nazwa[MAX_TOKEN];
    char typ;
} token;
```

Rozpoznawane są następujące typy tokenów:

```
#define PRAWY_NAWIAS 1
#define LEWY_NAWIAS 2
#define OPERATOR 3
#define TEKST 5
#define L_CALK 6
#define L_ZMP 7
#define KONIEC 10
```

Wśród operandów rozróżnia się teksty, liczby całkowite oraz zmiennoprzecinkowe. Zestaw ten jest wystarczający dla większości rodzajów wyrażeń. Zawartość tokena jest umieszczana w polu `nazwa` jako ciąg znaków. Jest to najbardziej uniwersalny sposób zapisu. Zdefiniowanie tego pola, jako tablicy znaków a nie wskaźnika na ciąg znaków, jest podyktowane tym, że liczba tokenów w wyrażeniach jest zwykle niewielka i dlatego posłużenie się wskaźnikami zmniejszyłoby w sposób nieistotny zapotrzebowanie na pamięć. Drugą przyczyną skłaniającą do użycia tablic jest fakt, że tokeny są często tworzone i kasowane, co w wyniku stosowania wskaźników może doprowadzić do zbytecznej segmentacji sterty (patrz p. 3.3).

Opis każdego ze stosowanych operatorów jest umieszczony w strukturze `operator`:

```
typedef struct
{
    char *nazwa;
    char typ;
```

```

    char priorytet;
    f_int f;
} operator;

```

Nazwa jest to ciąg znaków identyfikujący operator np. "sinus" w wyrażeniach arytmetycznych. Rozpoznawane są następujące typy operatorów:

```

#define UNARNY 8
#define BINARNY 9

```

Priorytet operatora wzrasta wraz z wartością pola `priorytet`. Typ `f_int` jest to wskaźnik na funkcję typu `int`: `typedef (*f_int)();`

Pole to określa funkcję stosowaną do obliczenia wartości definiowanego operatora. Do pobierania operandów i przekazywania wyników funkcja ta zwykle stosuje funkcje `ze_stosu` i `na_stos`. We wstępnej fazie działania funkcja ta powinna sprawdzić, czy jest dostępna właściwa liczba operandów i czy są one właściwego typu.

Opisy wszystkich operatorów są zgrupowane w tablicy. Ostatni jej element ma nazwę równą `NULL`. Dla omawianego tu języka tablica ta może przyjąć postać:

```

operator teksty[] =
{
    "*", BINARNY, 10, txt_ilocz,
    "+", BINARNY, 9, txt_suma,
    "dokl", UNARNY, 12, txt_dokl,
    "-", UNARNY, 11, txt_neg,
    NULL,
};

```

Najważniejszą funkcją modułu jest funkcja `zamien_na_onp`.

```

int zamien_na_onp(char **txt, operator *op, token *tab, int dl_tab)
{
    int dl_out, typ1, typ, jest_wolny, koniec_pw;
    token stos[MAX_STOS];
    char tok[MAX_TOKEN], tok1[MAX_TOKEN];
    memset(tab, 0, sizeof(token)*dl_tab);
    dl_out = jest_wolny = 0;
    do{
        switch((typ = daj_token(txt, tok, op)))
        {
            case TEKST: case L_CALK: case L_ZMP:
                WYPISZ(tab, dl_tab, typ, tok, dl_out)
                break;

```



```
case LEWY_NAWIAS:
    NA_STOS(stos,MAX_TOKEN,typ,tok,jest_wolny)
    break;
case PRAWY_NAWIAS:
    ZE_STOSU(stos,typ,tok,jest_wolny)
    while (typ!=LEWY_NAWIAS)
    {
        WYPISZ(tab,dl_tab,typ,tok,dl_out)
        ZE_STOSU(stos,typ,tok,jest_wolny)
    }
    break;
case OPERATOR:
    ao{
        if (jest_wolny==0)
            koniec_pw=TRUE;
        else if (stos[jest_wolny-1].typ==LEWY_NAWIAS)
            koniec_pw=TRUE;
        else if (pr(stos[jest_wolny-1].nazwa,op)<pr(tok,op))
            koniec_pw=TRUE;
        else
        {
            koniec_pw=FALSE;
            ZE_STOSU(stos,typ1,tok1,jest_wolny);
            WYPISZ(tab,dl_tab,typ1,tok1,dl_out);
        }
        } while (koniec_pw==FALSE);
    NA_STOS(stos,MAX_STOS,typ,tok,jest_wolny);
    break;
case ERR:
    return(ERR);
case KONIEC:
    while (jest_wolny>0)
    {
        ZE_STOSU(stos,typ1,tok1,jest_wolny);
        WYPISZ(tab,dl_tab,typ1,tok1,dl_out);
    }
    } while (typ!=KONIEC);
return(OK);
}
```

Jak widać, funkcja ta przepisuje bezpośrednio po wczytaniu wszystkie operandy do tablicy wyjściowej. W przypadku wczytania operatora lub nawiasu sposób działania jest bardziej złożony:

1. Nawias lewostronny jest dopisywany do stosu operatorów.
2. Nawias prawostronny powoduje przeniesienie ze stosu do tablicy wyjściowej wszystkich operandów, aż do napotkania nawiasu lewostronnego, który jest tylko usuwany ze stosu.
3. Przetworzenie całego wyrażenia powoduje przeniesienie ze stosu do wyjściowej tablicy wszystkich tam umieszczonych operandów.
4. Wczytany operator *op* jest zapisywany na stosie jeżeli:
 - stos jest pusty
 - na wierzchołku stosu jest nawias lewostronny
 - na wierzchołku stosu jest operator o priorytecie niższym od priorytetu operatora *op*.
5. Jeżeli nie zachodzi żaden z wymienionych warunków, to na wyjście kierowane są kolejno pobierane ze stosu operatory.

Makra *ZE_STOSU*, *NA_STOS* i *WYPISZ* wprowadzono celem zwiększenia czytelności kodu:

```
#define WYPISZ(tab,dl_tab,typ,tok,dl_tok) \
    if (na_wyjście(tab,dl_tab,typ,tok,&dl_out)==ERR) \
        return(ERR);
#define NA_STOS(stos,max,typ,tok,jest_wolny) \
    if (na_stos(stos,max,typ,tok,&jest_wolny)==ERR) \
        return(ERR);
#define ZE_STOSU(stos,typ,tok,wolny) \
    if (ze_stosu(stos,&typ,tok,&wolny)==ERR) \
        return(ERR);
```

Kolejne tokeny są pobierane z wejściowego ciągu przez funkcję *zamien_na_onp* za pomocą funkcji *daj_token*:

```
static int daj_token (char **txt, char *token, operator *op)
{
    while (**txt && isspace(**txt))
        (*txt)++;
    if (!**txt)
        return(KONIEC);
    if (isalpha(**txt) || **txt=='"')
        return(tekst_nazwa(txt,token,op));
    if (isdigit(**txt))
```

```

    return(wczytaj_liczby(txt,token));
return(operator_nawias(txt,token,op));
}

```

Funkcja `daj_token` pomija nieznaczące znaki niedrukowalne, rozdzielające tokeny. Litera rozpoczynająca token może oznaczać początek operatora lub tekst. Interpretacją ciągów tego typu zajmuje się funkcja `tekst_nazwa`. Jest ona też odpowiedzialna za połączenie w jeden ciąg znaków ujętych w apostrofy. Pozostałe typy tokenów są obsługiwane przez funkcje `wczytaj_liczby` (dotyczy to zarówno liczb całkowitych, jak i liczb zmiennoprzecinkowych) oraz funkcji `operator_nawias`. Ostatnia funkcja ma za zadanie identyfikowanie operatorów jednoznakowych.

Wszystkie te funkcje są skonstruowane w podobny sposób. Dla przykładu poniżej zamieszczono kod funkcji `tekst_nazwa`:

```

static int tekst_nazwa (char **txt, char *token, operator *op)
{
    int i;
    char *tok=token;
    if (**txt=='')
    {
        (*txt)++;
        for (i=0; i<MAX_TOKEN && **txt; ++i)
        {
            if (**txt=='')
                if (*(++(*txt))!='')
                {
                    *token=0;
                    return TEKST;
                }
            *token++=**(*txt)++;
        }
        if (i>MAX_TOKEN || **txt==0)
            return ERR;
        return OK;
    }
    for (i=0; i<MAX_TOKEN; ++i)
        if (**txt && (isalnum(**txt) || **txt=='_'))
            *token++=**(*txt)++;
        else
            break;
    if (i==MAX_TOKEN)

```

```

    return(ERR);
*token=0;
return ((to_jest_operator(tok,op)==TRUE)? OPERATOR: TEKST);
}

```

Zmienna `tok` została wprowadzona tylko po to, aby można było sprawdzić, czy utworzony ciąg jest nazwą operatora (za pomocą funkcji `to_jest_operator`). Wskaźnik `token` jest w trakcie działania funkcji inkrementowany. Tracona jest tym samym informacja o początkowej wartości tego wskaźnika, konieczna do wywołania funkcji `to_jest_operator`. Na uwagę zasługuje też sposób inkrementowania wskaźnika na znak i wskaźnika na wskaźnik na znak.

6.3. Przeszukiwanie tekstu

Opisanie tekstu pozwala użytkownikowi na określenia, jakie teksty są dla niego użyteczne. Do rozwiązania pozostał jeszcze ostatni problem, tj. obliczenie wartości wyrażeń elementarnych. Zwykle wymaga to przeszukiwania tekstów bazy. Podstawowy problem, z jakim mamy do czynienia, to zapewnienie utrzymania czasu przeszukiwania bazy w granicach pozwalających na pracę interakcyjną. Nie jest to łatwe zważywszy, iż baza taka może zawierać kilka megabajtów tekstu. Inny problem, z jakim mamy do czynienia, to ten, że nie zawsze użytkownik jest w stanie dokładnie określić postać interesującego go ciągu.

Proponowane rozwiązanie polega na stosowaniu algorytmów znajdowania poszukiwanego ciągu znaków, zwanego dalej wzorcem, wielokrotnie szybszych od najprostszych, stosowanych zwykle algorytmów a także stosowaniu zbioru inwersyjnego radykalnie przyspieszającego przeszukiwania wybranych wzorców. Drugi z tych problemów rozwiązany jest przez wprowadzenie funkcji ustalającej podobieństwo wyrazów.

6.3.1. Szybkie szukanie wzorca

Najprostszy sposób poszukiwania wzorcowego ciągu znaków w tekście polega na jego sekwencyjnym przeszukiwaniu. W jego trakcie porównywane są kolejne znaki tekstu i wzorca. Sposób ten wykorzystuje funkcja `wolne_szukanie`:

```

char *wolne_szukanie (char *c, char *w)
{
    int i, dlw;
    char *kon;
    dlw=strlen(w);
    kon=c+strlen(c);

```

```

while (c<kon)
{
    for (i=0; i<dlw && c[i]==w[i]; ++i);
        if (i==dlw)
            return(c);

    c++;
}
return(NULL);
}

```

Funkcja ta przyjmuje wartość równą wskaźnikowi na pierwsze wystąpienie wzorca lub *NULL*, gdy nie występuje on w tekście. Niezgodność tekstu i wzorca powoduje inkrementowanie wskaźnika *c*, po czym proces badania rozpoczyna się od początku wzorca. Taki, intuicyjnie oczywisty, sposób działania powoduje, że funkcja ta na znalezienie wzorca wymaga liczby porównań znaków proporcjonalnej średnio do $\text{strlen}(c) \cdot \text{strlen}(w)$. Znanych jest kilka algorytmów wymagających do tego celu znacznie mniejszej liczby porównań. Poniżej przedstawiony algorytm jest zmodyfikowana wersja algorytmu Boyera-Moore'a [ALGinC].

Algorytm ten do znalezienia wzorca wymaga około $\text{strlen}(c) + \text{strlen}(w)$ porównań znaków, w przypadku zaś, gdy alfabet z którego są utworzone teksty jest dość liczny a wzorce niezbyt długie, liczba ta zmniejsza się do około $\text{strlen}(c) / \text{strlen}(w)$. Tak znaczna redukcję liczby porównań znaków osiąga on dzięki wstępnej analizie wzorca oraz zmianie kierunku jego porównywania - jest on badany począwszy od końca.

Zasadę działania algorytmu ilustruje poniższy przykład. Załóżmy, że w poszukiwaniu wzorca "abb" natknęliśmy się w tekście na znak 'c'. Jak już to wspomniano, analiza odbywa się od strony prawej do lewej. Znak 'c' nie występuje we wzorcu, zatem w przeszukiwaniu tekstu możemy ominąć 3 znaki (długość wzorca) bez potrzeby ich badania. Takie samo działanie możemy podjąć naturalnie także po natrafieniu w tekście na jakikolwiek inny znak nie występujący we wzorcu.

Sytuacja komplikuje się, gdy spotkamy znak będący co prawda we wzorcu, ale występujący na niewłaściwej pozycji, np. znak 'a' na samym początku badania. Do wyznaczenia liczby znaków do ominięcia w takim wypadku wykorzystywana jest tablica *pomin*. Dla każdego ze znaków wzorca podaje ona indeks ostatniego jego wystąpienia we wzorcu, np. dla wzorca "abb" *pomin*['a']=0 zaś *pomin*['b']=2. Funkcja *ini_pomin* inicjuje dla podanego wzorca tablicę *pomin*. Dla znaków nie występujących w nim wpisywana jest wartość -1.

```
static char pomin['z'+1];

static void ini_pomin(char *w)
{
    int i;
    memset(pomin, 0xFF, sizeof(pomin));
    for (i=0; w[i]!=0; ++i)
        pomin[w[i]]=i;
}
```

Niech s będzie wskaźnikiem na znak w tekście, od którego potencjalnie może rozpocząć się wzorzec, zaś w wskaźnikiem na wzorzec. Zgodnie z wyżej wymienioną zasadą porównywanie rozpoczynamy od i -tego znaku, tzn. $s[i]$ oraz $w[i]$, gdzie $i = \text{strlen}(w) - 1$. Jeżeli znaki są sobie równe, to dekrementujemy i . Dojście do początku wzorca oznacza naturalnie jego znalezienie. Niezgodność tekstu i wzorca możliwa jest w dwu przypadkach. Znak $s[i]$ może być znaleziony "za wcześnie" ($i > \text{pomin}[s[i]]$) i wtedy należy zwiększyć s o wartość $(i - \text{pomin}[s[i]])$ lub też "za późno" ($i < \text{pomin}[w[i]]$) i wtedy s należy zwiększyć o $(\text{pomin}[w[i]] - i)$.

Szybkie poszukiwanie wzorca realizuje funkcja **szybkie_szukanie**:

```
char *szybkie_szukanie(char *c, char *w)
{
    int i, dlw, krok;
    char *kon;
    dlw = strlen(w);
    kon = c + strlen(c);
    ini_pomin(w);
    while (c < kon)
    {
        for (i = dlw - 1; i < dlw && c[i] == w[i]; --i);
        if (i < 0)
            return(c);
        krok = pomin[c[i]];
        c += (i > krok) ? i - krok : krok - i + 1;
    }
    return(NULL);
}
```

Jak widać, powyższa funkcja różni się od funkcji **wolne_szukanie** jedynie inicjowaniem tablicy **pomin**, zmianą kierunku badania i sposobem zwiększania wskaźnika c .

6.3.2. Tablica inwersyjna

Jeżeli baza danych zawiera kilka milionów znaków tekstu, to nawet najszybsze komputery i algorytmy sekwencyjnego przeszukiwania danych nie są w stanie zapewnić akceptowalnego czasu oczekiwania na odpowiedź. Radykalne przyspieszenie działania oferują dopiero tablice inwersyjne. W tablicy takiej umieszczone są wybrane terminy wraz z identyfikatorami tekstów, w jakich one wystąpiły. Jeżeli poszukiwany ciąg jest umieszczony w tablicy, to zamiast przeglądania całej bazy wystarcza sięgnięcie do odpowiedniego elementu tablicy inwersyjnej.

Poniżej przedstawiono prostą implementację tablicy inwersyjnej. Każdy termin umieszczony w tablicy jest zapamiętany w pliku wraz z wykazem przypisanych mu tekstów. W celu przyspieszenia pracy, po otwarciu tablicy, wszystkie terminy (bez wykazu tekstów) są kopiowane do pamięci operacyjnej. Liczba terminów nie przekracza zwykle kilkuset, zatem można założyć, że użytkownik dysponuje wystarczającą wielkością pamięci na ich zapisanie.

6.3.2.1. Omówienie funkcji

Przedstawione tu funkcje tworzą warstwę *inwers*. Podczas jej tworzenia skorzystano z funkcji warstwy *dzd*, wszystkie odwołania do plików są wykonane za pomocą funkcji *dzd*. Definicje wykorzystywanych stałych, struktur wraz z deklaracjami omawianych poniżej funkcji są podane w pliku nagłówkowym *inwers.h*. Aby zaznajomić się ze sposobem działania poniższych funkcji, konieczne jest poznanie struktury *ops_ter*:

```
typedef struct
```

```
{  
    char *termin;      /* wskaźnik na termin indeksowy */  
    long ident;        /* identyfikator z terminu w bazie dzd */  
    unsigned *dane;    /* tablica identyfikatorów tekstów */  
    int dl_danych;     /* długość tablicy identyfikatorów */  
} ops_ter;
```

Identyfikator musi być nadany terminowi przez funkcję *inv_ins*, pozostałe pola są wypełniane przez użytkownika lub standardowe funkcje omawianej warstwy. Jak widać, tablica identyfikatorów tekstów jest utworzona z liczb typu *unsigned*. Pozwala to na skrócenie prawie o połowę rozmiarów tej tablicy (w porównaniu z identyfikatorami typu *long*), a jednocześnie umożliwia systemowi korzystanie z 65536 różnych tekstów, co jest wielkością wystarczającą w większości przypadków.

Wszystkie funkcje tworzące warstwę *inwers* są typu *int*. Poprawne wykonanie powoduje przyjęcie przez nią wartości *OK*, a błędne - *ERR*. Stałe te są zdefiniowane w pliku nagłówkowym *dzd.h*. Omawiane tu funkcje można podzielić na trzy grupy. Wykonują one działania na: całej tablicy, całym elemencie tablicy oraz na pojedynczym jej elemencie.

Pierwszą grupę tworzą funkcje: *inv_ini*, *inv_open*, *inv_close*:

```
int inv_ini (nazwa_pl, terminy, inv)
/* zakładanie nowej tablicy inwersyjnej */
char *nawa_pl; /* nazwa tworzonej tablicy inwersyjnej */
char *terminy[]; /* terminy do umieszczenia w tablicy */
ops_inv *inv; /* wskaźnik na strukturę opisującą tablicę */
```

Nazwa tablicy podlega takim samym ograniczeniom jak nazwa bazy, tj. musi to być pozbawiona rozszerzenia nazwa pliku w stosowanym systemie operacyjnym. Jeżeli już istnieje tablica o takiej samej nazwie, to jej zawartość jest kasowana. Wykaz terminów przewidzianych do zapamiętania w bazie jest przekazywany jako drugi parametr funkcji. Koniec wykazu sygnalizuje wskaźnik o wartości *NULL* lub wskaźnik na ciąg pusty, tj. *""*. Wszystkie informacje o tworzonej liście są zapamiętywane w strukturze **inv*.

Każdemu z terminów przypisana jest pusta lista identyfikatorów. Dopisywanie do niej identyfikatorów tekstów, w których dany termin występuje, odbywa się za pomocą funkcji *inv_add*.

Wykaz terminów przewidzianych do umieszczenia w tablicy inwersyjnej jest zwykle znany przed rozpoczęciem indeksowania tekstów. Często określa się go na podstawie analizy słownictwa przeprowadzanej przez człowieka lub komputer, bądź też korzysta się z terminów zaczerpniętych z opracowanych już tezaurusów. Tezaurusem nazywamy słownik zawierający wykaz terminów wraz z relacjami zachodzącymi między nimi. Do najczęściej stosowanych relacji należą: "stosuj w zamian", "termin węższy", "termin szerszy".

```
int inv_open (nazwa_pl, inv)
/* otwieranie istniejącej tablicy inwersyjnej */
char *nawa_pl; /* nazwa tworzonej tablicy inwersyjnej */
ops_inv *inv; /* wskaźnik na strukturę opisującą tablicę */
```

Otwierana funkcja *inv_open* tablica musi istnieć. Wszystkie dane o otwieranej liście są umieszczane w strukturze **inv*.

```
int inv_close (inv)
/* zamykanie używanej tablicy inwersyjnej */
ops_inv *inv; /* wskaźnik na strukturę opisującą tablicę */
```


Funkcja `inv_close` zamyka tablicę inwersyjną, korzystanie z tablicy jest możliwe dopiero po jej ponownym otwarciu.

Funkcje drugiej grupy realizują podstawowe operacje na całych elementach tablicy inwersyjnej: wpisywanie (`inv_ins`), wczytywanie (`inv_get`), kasowanie (`inv_del`) oraz zmiana zawartości elementów (`inv_rep`).

```
int inv_ins (inv, dane)
                /* dopisanie nowego terminu do listy inwersyjnej */
ops_inv *inv;   /* wskaźnik na strukturę opisującą tablicę */
ops_ter *dane;  /* wskaźnik na strukturę opisującą element */
```

Funkcja wykona swe zadanie, jeżeli podany element nie został już wpisany do tablicy i dysponujemy wystarczającą ilością pamięci na jego dopisanie. Przed wywołaniem `inv_ins` należy w strukturze `*dane` wypełnić pola `termin`, `dane` i `dl_danych`. Funkcja nadaje terminowi identyfikator w bazie `dzd` i wpisuje go do pola `dane->ident`.

```
int inv_get (inv, dane)
                /* pobranie elementu tablicy inwersyjnej */
ops_inv *inv;   /* wskaźnik na strukturę opisującą tablicę */
ops_ter *dane;  /* wskaźnik na strukturę opisującą element */
```

Nazwa poszukiwanego terminu jest wpisywana do pola o nazwie `termin` w strukturze `*dane`. Jeżeli termin został wcześniej zapisany w tablicy `*inv`, to tablica związanych z nim identyfikatorów jest umieszczana w strukturze `*dane`.

```
int inv_del (inv, dane)
                /* kasowanie terminu umieszczonego
                w tablicy inwersyjnej */
ops_inv *inv;   /* wskaźnik na strukturę opisującą tablicę */
char *termin;   /* nazwa dopisywanego elementu */
```

Funkcja `inv_del` kasuje termin wprowadzony uprzednio do tablicy.

```
int inv_rep (inv, dane)
                /* zmiana treści elementu tablicy inwersyjnej */
ops_inv *inv;   /* wskaźnik na strukturę opisującą tablicę */
ops_ter *dane;  /* wskaźnik na strukturę opisującą element */
```

Funkcja `inv_rep` zmienia zawartość tablicy przypisaną terminowi o nazwie `dane->termin`. Próba modyfikacji w ten sposób, terminu nie zapisanego w tablicy inwersyjnej, powoduje wystąpienie błędu.

Kolejna, ostatnia grupa funkcji, odpowiedzialna jest za zmianę zawartości tablicy przypisanej jednemu terminowi.

```
int inv_add(inv, termin, dopisz);
        /* dopisanie nowego identyfikatora
           do elementu tablicy inv */
ops_inv *inv;    /* wskaźnik na strukturę opisującą tablicę */
char *termin;    /* nazwa dopisywanego elementu */
unsigned dopisz; /* dodawany identyfikator tekstu */
```

Funkcja `inv_add` uzupełnia tablicę identyfikatorów tekstów o nowy element. Warunkiem poprawnego zakończenia działania funkcji jest uprzednie zapisanie elementu `termin` do tablicy inwersyjnej oraz niewystępowanie na liście przypisanej temu terminowi wartości `dopisz`.

6.3.2.2. Analiza kodu funkcji

Oprócz omówionej już struktury `ops_ter` warstwa `inwers` korzysta ze struktury `ops_inv`. Zapisywane w niej są wszystkie informacje o każdej ze stosowanych tablic inwersyjnych.

```
typedef struct
(
    ops_dzd baza;
    typ_ter *dane;
    unsigned liczba_term;
) ops_inv;
```

Pole `baza` przeznaczone jest na strukturę z opisem bazy danych zmiennej długości. Stosowana jest ona do przechowywania list identyfikatorów tekstów. Wskaźnik `dane` wskazuje na początek ciągu opisów terminów tablicy inwersyjnej. Liczba terminów w liście jest zapisana w pliku jako pierwszy rekord bazy `dzd`. Dodatkowo po otwarciu bazy, aktualna wartość tej liczby jest przechowywana w polu `liczba_term`. Z liczby tej korzysta się wielokrotnie, zapamiętanie jej w pamięci operacyjnej pozwala na skrócenie czasu przetwarzania.

```
typedef struct
(
    char *termin;
    int iCent;
) typ_ter;
```

Pole `ident` zawiera identyfikator rekordu w bazie `dzd`, przypisanego danemu terminowi. Rekord ten zawiera termin wraz z listą identyfikatorów tekstów. Liczba terminów nie przekracza zwykle tysiąca i dlatego też identyfikatory są typu `int`.

Poniżej omówiony zostanie kod trzech funkcji `inv_ini`, `inv_open` oraz `inv_get`:

```
int inv_ini(char *nazwa_pl, char *terminy[], ops_inv *inv)
{
    long pocz_ident;
    ops_ter o;
    dzd d;
    unsigned i;
    if (dzd_ini(&inv->baza,100,nazwa_pl)==ERR)
        return(ERR);
    for (i=0; terminy[i]!=NULL && *terminy[i]!=0; ++i);
        inv->liczba_term=++i;
    o.termin=strdup(licznosc);
    o.dl_danych=sizeof(unsigned);
    o.dane=malloc(sizeof(unsigned));
    *o.dane=i;
    if (ter_dzd(&d,&o)!=OK)
        return(ERR);
    if ((pocz_ident=dzd_ins(&inv->baza,d.dl_txt,d.txt))==ERR)
        return(ERR);
    inv->dane[0].termin=o.termin;
    inv->dane[0].ident=pocz_ident;
    free(d.txt);
    free(o.dane);
    for (i=0; i<inv->liczba_term-1; ++i)
    {
        inv->dane[i+1].ident=
            dzd_ins(&inv->baza,strlen(terminy[i])+1,terminy[i]);
        if ((inv->dane[i+1].termin=strdup(terminy[i]))==NULL)
            return(ERR);
    }
    qsort(inv->dane,inv->liczba_term,sizeof(typ_ter),por_dane);
    return(OK);
}
```

Funkcja `inv_ini` rozpoczyna pracę od założenia bazy rekordów zmiennej długości. Wybrana wielkość bloku pozwala na zapamiętanie w jednym bloku 50 identyfikatorów tekstów. Następnie jest ustalana liczba terminów przewidzianych do wpisania. Jest ona większa o jeden od liczby terminów dostarczonych przez użytkownika. Pierwszy z nich podaje liczbę terminów w bazie. Jego nazwa jest podana w tablicy `licznosc`:

```
char licznosc[]="@lt@";
```

Liczba ta jest wykorzystana podczas alokowania pamięci na tablicę opisu terminów. Podczas otwierania istniejącej bazy możemy dzięki temu określić rozmiary tablicy na przechowywanie opisów terminów, bez potrzeby przeczytania całego pliku ją zawierającego. Funkcja `ter_dzd` zapełnia strukturę typu `dzd` na podstawie danych zapisanych w strukturze `ops_ter`.

```
static int ter_dzd(dzd *d, ops_ter *o)
{
    int dl_ter;
    dl_ter=strlen(o->termin)+1;
    d->ident=o->ident;
    d->dl_txt=o->dl_danych+dl_ter;
    if ((d->txt=malloc(d->dl_txt))==NULL)
        return(ERR);
    strcpy(d->txt,o->termin);
    if (o->dl_danych>0)
        memmove(d->txt+dl_ter,o->dane,o->dl_danych);
    return(OK);
}
```

Funkcja ta ułatwia warstwie `inwers` korzystanie z funkcji warstwy `dzd`.

Po zapisaniu pierwszego, "technicznego" rekordu do tablicy, wpisywane są tam kolejne ciągi z tablicy `termin`. Wpisywanie ich jest łatwiejsze, ich treścią jest bowiem jedynie sam termin. Są one zarazem umieszczane w tablicy opisu terminów, przechowywanej w strukturze `*inv`. Na zakończenie funkcja sortuje tę tablicę w kolejności alfabetycznej terminów.

Funkcja `inv_open` przygotowuje do użycia już istniejącą tablicę inwersyjną:

```
int inv_open(char *nazwa_pl, ops_inv *inv)
{
    dzd d;
    int i;
    if (dzd_open(&inv->baza,nazwa_pl)==ERR)
        return(ERR);
    if (dzd_fst(&inv->baza)==ERR)
        return(ERR);
    dzd_get(&inv->baza,&d);
    if (strcmp(d.txt,licznosc)!=0)
```

```

    return(ERR);
    memmove(d.txt+strlen(d.txt)+1,
    &inv->liczba_term,sizeof(unsigned));
    if ((inv->dane=malloc(inv->liczba_term*sizeof(typ_ter)))==NULL)
        return(ERR);
    for (i=0; i<inv->liczba_term; ++i)
    {
        dzd_get(&inv->baza,&d);
        if ((inv->dane[i].termin=strdup(d.txt))==NULL)
            return(ERR);
        inv->dane[i].ident=d.ident;
    }
    qsort(inv->dane,inv->liczba_term,sizeof(typ_ter),por_dane);
    return(OK);
}

```

Funkcja ta otwiera żadaną bazę dzd i wczytuje pierwszy jej rekord. Określa on liczbę terminów w zapisanych w bazie. Pozwala to na ustalenie wielkości tablicy w pamięci operacyjnej, potrzebnej do ich zapamiętania. Tablica ta jest następnie sortowana według kolejności alfabetycznej.

Ostatnią z omawianych funkcji jest funkcja *inv_get*, pobierająca opis terminu:

```

int inv_get(ops_inv *inv, ops_ter *dane)
{
    typ_ter szukam, *wyn;
    char *wyraz;
    dzd lista;
    wyraz=szukam.termin=dane->termin;
    if ((wyn=bsearch(&szukam,inv->dane,inv->liczba_term,
        sizeof(typ_ter),por_dane))==NULL)
        return(ERR);
    inv->baza.akt_ident=dane->ident=wyn->ident;
    if (dzd_get(&inv->baza,&lista)==ERR)
        return(ERR);
    if (strcmp(wyraz,lista.txt)!=0)
        return(ERR);
    dane->dl_danych=lista.dl_txt-strlen(wyraz)-1;
    if (dane->dl_danych==0)
    {
        dane->dane=NULL;
        return(OK);
    }
}

```

```
    }  
    if ((dane->dane=malloc(dane->dl_danych))==NULL)  
        return(ERR);  
    memmove(dane->dane, lista.txt+strlen(wyraz)+1, dane->dl_danych);  
    return(OK);  
}
```

Do szukania terminu w tablicy `inv->dane` wykorzystano szukanie połówkowe, tablica ta jest bowiem zawsze posortowana. Po wczytaniu odpowiedniego rekordu kontrolowane jest, czy zaczyna się on od poszukiwanego terminu. Brak zgodności oznacza istnienie błędów w bazie i konieczność przeindeksowania tekstów. Terminy bez listy identyfikatorów tekstów są traktowane w inny sposób niż wszystkie pozostałe. Wynika to z nieokreślonego działania funkcji `malloc` podczas próby alokowania obszaru o długości 0 bajtów.

Praktyczne wykorzystanie tablicy inwersyjnej możliwe jest po dopisaniu funkcji służących np. do usuwania identyfikatora tekstu z listy, wykonywania operacji wyznaczania sumy i iloczynu list, analizy tekstów w celu zidentyfikowania terminów z listy, itp. Napisanie tych funkcji pozostawiamy Czytelnikowi jako ćwiczenie.

Proces zakładania tablic inwersyjnych jest bardzo czasochłonny, a modyfikacja tekstu wymusza ich aktualizację. W wielu systemach są to jednak czynności wykonywane sporadycznie. Właściwy dobór terminów umieszczonych na liście sprawia, że czas obsługi listy wzrasta nieznacznie wraz ze zwiększaniem się liczby tekstów.

6.3.3. Określanie podobieństwa ciągów znaków

W dotychczasowych rozważaniach zakładaliśmy, że użytkownik potrafi dokładnie podać postać interesującego go ciągu znaków. Jest to założenie w dużej mierze nierealistyczne z dwu powodów.

Pierwszym z nich jest silnie rozwinięta fleksja niektórych języków naturalnych, w tym również języka polskiego. Jeżeli użytkownik poda w wyrażeniu termin "komputer", to ze znaczną dozą prawdopodobieństwa można założyć, że interesują go także teksty zawierające słowa: komputerom, komputery, skomputeryzowane itp. Często spotykane w języku polskim wymiany wewnątrztematyczne zmniejszają efektywność prostych algorytmów w usuwaniu końcówek i sprawiają, że stopa błędów osiągana w wyniku automatycznej identyfikacji tematu słowa, uzyskiwana nawet przez dość skomplikowane algorytmy, wynosi zwykle około 10%.

Drugi z tych powodów wynika z faktu, że zwykle nie jesteśmy w stanie zapewnić poprawności ortograficznej dużych baz tekstowych. Wprowadzane dane, jak i pytania, nie podlegają rygorystycznej kontroli i dobrze by było, gdyby system uwzględnił podobieństwo słów "komputer" i "komptuer".

Znanych jest wiele metod okreslania podobieństwa słów, przegląd ich jest zamieszczony w pracy [ALGinC]. Poniżej zaprezentowana metoda opiera się na wyznaczaniu liczby współwystępujących w badanych słowach n-gramów, to jest ciągów o długości równej n znaków.

Algorytm jest przeznaczony do analizy słów i dlatego możliwe jest wyznaczenie maksymalnej długości ciągu:

```
#define MAX_SLOWO 40
```

Obliczanie podobieństwa realizuje funkcja podob:

```
int podob (s1, s2, dln)
char *s1;      /* wskaźnik na pierwsze analizowane słowo */
char *s2;      /* wskaźnik na drugie analizowane słowo */
int dln;       /* długość n-gramu */
{
    unsigned dlc1, dlc2, i;
    unsigned razem, wspolne;
    if ((dlc1=strlen(s1))<dln || (dlc2=strlen(s2))<dln ||
        dlc1>MAX_SLOWO || dlc2>MAX_SLOWO)
        return(0);
    for (i=dln, wspolne=0; i<=dlc1; s1++, i++)
        wspolne+=liczba_wystapien(s1,s2,dln);
    razem=dlc1+dlc2-(dln<(1)+2;
    return((wspolne<(8)/razem);
}
```

Funkcja przyjmuje wartość z przedziału od 0 (podobieństwo minimalne) do 128 - podobieństwo maksymalne. Przyjęcie takiego przedziału pozwala na zastąpienie czasochłonnego dzielenia operacją przesuwania bitów podczas obliczania wartości funkcji. Funkcja `liczba_wystapien(s1,s2,dln)` podaje ile razy ciąg utworzony z dln początkowych znaków s1 występuje w ciągu s2:

```
static int liczba_wystapien (w, slowo, dln)
char *w;       /* ciąg zawierający n-gram */
char *slowo;   /* badane słowo */
int dln;       /* długość n-gramu */
{
    char *s=slowo;
```

```
char wzor[MAX_SLOWO+1];
int wynik=0;
memset(wzor,0,sizeof(wzor));
strncpy(wzor,w,dln);
while ((s=strstr(s,wzor))!=NULL)
    wynik++, s++;
return(wynik);
}
```

Jak widać, do przeszukiwania ciągu wykorzystano standardową funkcję *strstr* w miejsce wprowadzonej uprzednio *szybkie_szukanie*. Jest to spowodowane tym, że badane n-gramy mają niewielką długość, nie przekraczającą zwykle 4 znaków, i czynności wstępne – konieczne na rozpoczęcie badania za pomocą funkcji *szybkie_szukanie* – niwelują jej dużą szybkość działania.

Stosowane długości n-gramu wynoszą od 2 do 5, a graniczna wartość podobieństwa przekracza zwykle 60. Dobre rezultaty uzyskuje się dla 3-znakowych n-gramów i wartości granicznej równej 64. Analizę sposobu jej określania w zależności od długości słowa znaleźć można w pracy [ALGinC].

6.3.4. Podsumowanie

Do przeszukiwania danych tekstowych o rozmiarach przekraczających kilkadziesiąt kilobajtów konieczne jest stosowanie tablic inwersyjnych. Wskazówki na temat selekcji terminów zapisywanych w indeksie, w tym także sposobów automatycznego ich wyznaczania, znaleźć można w pracach poświęconych teorii wyszukiwania informacji. Tam też opisano tezaury – słowniki, w których określono relacje zachodzące między umieszczonymi w nich terminami. Pozwalają one na podniesienie zarówno selektywności, jak i kompletności wyszukiwanych tekstów.

Podczas tworzenia listy tekstów, odpowiadających danemu terminowi, dogodne jest umieszczanie w niej wszystkich tych tekstów, w których występuje słowo dostatecznie podobne do tego terminu. Funkcję wyznaczającą podobieństwo słów stosować można wtedy zarówno do wyszukiwania pełnotekstowego, jak i określania terminów z listy inwersyjnej, reprezentujących słowo podane w pytaniu. Jeżeli tylko stopień podobieństwa terminu z tablicy i słowa z pytania jest dostatecznie duży, to wynikające z tego powodu błędne zakwalifikowanie lub odrzucenie tekstów nie jest w praktyce istotne.

Lista terminów na liście jest ograniczona. Jeżeli użytkownik zleca wyszukiwanie ciągu znaków w niej nie umieszczonego, to zaleca się stosowanie funkcji *szybkie_szukanie*.

Rozdział 7

OPROGRAMOWANIE WSPOMAGAJĄCE

Pierwsi projektanci komputerów byli zarazem programistami. Z biegiem czasu nie tylko dokonał się podział na ludzi zajmujących się sprzętem i oprogramowaniem, ale wśród tych ostatnich wyodrębnili się projektanci i programiści. Stopień złożoności tworzonego oprogramowania jest obecnie tak wielki, iż nie sposób uzyskać profesjonalną jakość oprogramowania bez stosowania odpowiednich narzędzi. Dlatego też obecnie od programisty wymaga się nie tylko umiejętności zapisania danego algorytmu w wybranym języku programowania, ale również znajomości licznych programów wspomagających. Projektowanie dużego programu zbliżone jest złożonością do projektowania wielkiej budowli. Samodzielne kontrolowanie wszystkich jego elementów jest prawie niemożliwe. Ten rozdział przedstawia niektóre z narzędzi stosowanych do tworzenia dużych systemów. Ułatwiają one i częściowo automatyzują pracę programisty. Najczęściej stosowane spośród nich są: program zarządzający kompilacją wieloplikowych programów *Make*, program tworzący biblioteki funkcji *Lib*, program do wyszukiwania podanego ciągu w tekście *Grep*, program testujący *Profiler*, wyspecjalizowany edytor do pisania programów (np. *Brief*) i program wspomagający znajdowanie błędów *Debugger*.

7.1. Porównanie edytora programowego z edytorem ogólnego przeznaczenia

Już na samym początku tworzenia programu ważną rolę odgrywa edytor, za pomocą którego powstaje kod źródłowy aplikacji. Programista ma ogromny wybór wśród kilkunastu najczęściej stosowanych edytorów. Do najpopularniejszych należą: *Brief*, *Multiedit*, *Norton Editor*, *Kedit*, wszelkie odmiany unix-owego *vi* oraz stosowane w środowiskach zintegrowanych np. *PWB* edytor. Różnią się one organizacją ekranu i możliwościami.

Z punktu widzenia programisty najważniejszymi cechami różniącymi edytor programowy od edytora ogólnego przeznaczenia są:

1. Wielookienkowość, tj. umożliwienie jednoczesnego edytowania kilku plików na jednym ekranie w odrębnych oknach, jak i jednego pliku w kilku oknach. Wielkość tych okien można zmieniać. Jest to cecha szczególnie przydatna w programowaniu w języku C, kod źródłowy jest bowiem zapisany w wielu stosunkowo niewielkich plikach i jednoczesne poprawianie na ekranie kilku powiązanych ze sobą plików znacznie ułatwia programowanie, np. skopiowanie do plików nagłówkowych deklaracji funkcji, kontrolowanie poprawności ich użycia, sprawdzanie pisowni stałych itp.
2. Wspomaganie językowe. Pozwala ono na dostosowanie trybu pracy edytora do syntaktyki języka programowania. Obejmuje np. automatyczne generowanie wcięć po instrukcjach blokowych, dopisywanie nawiasów, ułatwienie wprowadzania słów kluczowych i wykrywanie błędów syntaktycznych. Pozwala to na automatyczne utrzymanie jednolitego stylu programowania, co upraszcza ich późniejszą modyfikację.
3. Sprawność manualna, tj. łatwość posługiwania się edytorem (minimalizacja liczby uderzeń w klawiaturę w celu uzyskania podstawowych operacji) oraz możliwość przededefiniowywania sposobu obsługi edytora zgodnie z nawykami użytkownika.

Zaden z edytorów ogólnego przeznaczenia nie daje wymienionych możliwości. Dlatego też proponujemy stosowanie edytorów wyspecjalizowanych. Wykaz najważniejszych komend w dwóch spośród nich, a mianowicie Brief'a 3.0 i edytora ze zintegrowanego środowiska Borland C++ 2.0, jest zamieszczony w załączniku (patrz p. 8.5).

Edytory te są znacznie bardziej skomplikowane i przydatne dla programisty niż to może wynikać z tabeli komend. W pakiecie edytora Brief znajduje się kompilator dla makr pisanych w języku C-podobnym. W pakiecie Borland C++ można używać myszy do operacji okienkowych (standard Microsoft) i do korzystania z menu. Dodatkowo o przydatności świadczy cena, szybkość działania i dobrze opracowany kontekstowy pomocnik wraz ze staranną dokumentacją użytkownika.

7.2. Zarządzanie programem

Jak już wielokrotnie zaznaczano, kod źródłowy oprogramowania systemu mieści się w kilku bądź nawet kilkudziesięciu plikach tekstowych. Zapisanie całości kodu w jednym pliku jest niecelowe, bowiem kłopotliwe jest jego poprawianie. Nawet najmniejsza poprawka wymaga powtórnej kompilacji całego oprogramowania, co jest procesem czasochłonnym. Rozwiązaniem tego typu problemów jest podzielenie

takiego pliku na kilka mniejszych części - i kompilowanie tylko tych, które uległy modyfikacji - i potem łączenie wszystkich modułów programem łączącym.

Programista może łatwo popełnić pomyłkę, np. gdy samodzielnie wybiera pliki do kompilacji. Możliwe jest jednak zautomatyzowanie tego procesu przez analizę daty i czasu powstawania plików. Jeśli np. plik źródłowy .C jest młodszy, czyli ma datę utworzenia późniejszą niż jego skompilowany moduł .OBJ, to znaczy że zmieniono go już po jego kompilacji i wobec tego konieczna jest jego powtórna kompilacja. Taka sama zależność dotyczy zbioru wynikowego .EXE i .OBJ, czyli procesu łączenia. Tak długo jak plik .EXE jest młodszy od wszystkich tworzących go plików .OBJ, łączenie nie jest wykonywane. W przeciwnym przypadku odpowiednie zbiory są powtórnie łączone. Dodatkowym utrudnieniem jest konieczność uwzględnienia zmian dokonywanych w plikach nagłówkowych. Zmiany w jednym z nich oznaczają, że konieczne jest skompilowanie wszystkich plików programowych korzystających z niego.

Automatyzacja procesu rekompilacji dla dużych programów złożonych z wielu plików źródłowych może odbywać się na dwa sposoby. Pierwszy z nich to używanie opcji **Project** lub **Make** w zintegrowanych środowiskach kompilatorów. Drugi wykorzystuje samodzielny program o nazwie **Make**. W obu przypadkach sposób postępowania jest podobny. Wieloplikowe programy czasami nazywa się projektami lub zbiorami plików. Każdy taki projekt jest opisany w oddzielnym pliku. Najczęściej spotykane jego rozszerzenia to .PRJ lub .MAK. Podczas definicji takiego projektu podaje się nazwy plików składowych. Po wybraniu opcji **Run** lub **Make** następuje kolejno kompilacja każdego zmienionego składnika i łączenie ich w jeden wykonywalny program. W opcji **Run** oprócz tego program zostanie uruchomiony.

Wieloplikowe programy używają nawet kilkudziesięciu plików nagłówkowych. Występują w nich definicje stałych, deklaracje zmiennych i typów. Każda zmiana w zbiorze nagłówkowym wymusza rekompilację zbiorów dołączających ten plik, więc i ta zależność powinna zostać uwzględniona w projekcie. Zależność tę zaznacza się np. przez umieszczanie w nawiasach okrągłych wszystkich wczytywanych za pomocą dyrektywy preprocesora (**#include**) zbiorów nagłówkowych.

Podczas tworzenia rozbudowanych systemów bardziej użyteczny jest program **Make**. Daje on bowiem większą swobodę w określaniu wykonywanych operacji i pozwala na uzyskanie większego kodu wynikowego niż jest to możliwe w środowisku zintegrowanym. Plik .MAK zawiera listę plików

głównych, zależnych i komendy użytkownika. Do każdego pliku głównego podana jest lista plików zależnych. Standardowy wygląd jest następujący:

plik główny 1 : lista zbiorów zależnych

komenda 1

komenda 2

plik główny 2 : lista zbiorów zależnych

komenda 3

...

plik główny N : lista zbiorów zależnych

komenda M

Nazwa zbioru głównego rozpoczyna się obowiązkowo w pierwszej kolumnie i jest oddzielona od listy zbiorów zależnych dwukropkiem. Wiersz komend musi być poprzedzony co najmniej jedną spacją lub znakiem tabulacji. Należy przy tym zwrócić szczególną uwagę na porządek podawania plików głównych, ponieważ program **Make** przetwarza kolejno wiersz po wierszu. Jako zasadę należy więc przyjąć, że najpierw podaje się zbiory nagłówkowe, później zaś źródłowe. Jako przykład nieprawidłowej konstrukcji niech posłuży poniższy plik definicyjny:

```
test.obj: test.c
```

```
    bcc test test1.obj
```

```
test1.obj: test1.c
```

```
    bcc -etest test1 test.obj
```

```
test.exe: test.h
```

```
    bcc -c test1
```

```
    bcc test test1.obj
```

Analiza sposobu działania programu **Make** wykazuje, że w przypadku modyfikacji zbioru nagłówkowego i któregośkolwiek ze zbiorów **.C** proces kompilacji będzie przebiegać błędną ścieżką. Najpierw wykona się kompilacja zbioru **.C** i utworzy się nowy zbiór **.EXE**. Wobec tego data utworzenia zbioru **.EXE** jest późniejsza od daty założenia zbioru **.H** i już nie wykona się powtórna kompilacja uzależnionego od **.H** drugiego zbioru źródłowego. Poprawić to można dokonując zmiany kolejności w następujący sposób:

```
test.exe: test.h
```

```
    bcc -c test1
```

```
    bcc test test1.obj
```

```
test.obj: test.c
```

```
    bcc test test1.obj
```

```
test1.obj: test1.c
    bcc -etest test1 test.obj
```

W niektórych sytuacjach, takich jak awaria zegara systemowego lub zmiana modelu pamięci, konieczne jest bezwarunkowe skompilowanie wszystkich plików tworzących projekt. W środowisku zintegrowanym służy do tego opcja **BUILD ALL**, poza nim zaś użycie programu **Make** musi zostać poprzedzone wywołaniem programu **Touch**, uaktualniającego datę i czas powstania wymienionych w linii zlecenia plików. Użyteczny jest też program **Prj2mak**, który przekształca zbiór typu **.PRJ** w zbiór **.MAK**.

Program **Nmake**, znajdujący się w pakiecie Microsoft C 6.0, można też wykorzystać do innych zadań, np. do składowania zaktualizowanych plików, czyli tych co mają nowszą datę i czas od tych znajdujących się w podanej kartotece.

```
# lista dodatkowo aktualizowanych rozszerzeń
.SUFFIXES: .h
# nazwa docelowego napędu i kartoteki
DIR = a:\
# schemat działania dla każdego rozszerzenia
.c($(DIR)).c:
    !COPY *** $(DIR)
.h($(DIR)).h:
    !COPY *** $(DIR)
# lista aktualizowanych zbiorów
UPDATE: $(DIR)\*.c $(DIR)\*.h
```

Definicje wykorzystanych symboli są następujące:

1. Znak # rozpoczyna wiersz komentarza.
2. **.SUFFIXES** jest listą rozszerzeń dla **Nmake**. Stosowany jest on, gdy tworzone są zbiory docelowe, dla których nie zostały wyspecyfikowane zbiory uzależnione. W razie niezalezienia zbiorów zależnych **Nmake** przeszukuje bieżącą kartotekę w poszukiwaniu zbioru o tej samej nazwie co zbiór docelowy z rozszerzeniem z listy **.SUFFIXES**. Domyślnie lista jest predefiniowana jak następuje:
.SUFFIXES: .obj .exe .c .asm. Dodanie nowego rozszerzenia następuje po napisaniu po słowie **.SUFFIXES:** tego rozszerzenia. Skasowanie listy następuje przez zdefiniowanie pustej listy: **.SUFFIXES:**.
3. **DIR** jest makrem użytkownika zdefiniowanym według schematu **nazwa_makra=ciąg** i automatycznie podstawiane jest w miejsce wywołania, czyli tam gdzie występuje **\$(nazwa_makra)**.

4. UPDATE zastępuje zbiór główny i służy do wyspecyfikowania listy zbiorów zależnych.
5. `.c$(DIR).c:` jak i `.h$(DIR).h:` są zdefiniowanymi regułami działania. `.c` i `.h` oznaczają rozszerzenia zbiorów zależnych, natomiast `$(DIR).c:` jak i odpowiednio `$(DIR).h:` - rozszerzenia zbiorów głównych. W nawiasach klamrowych `{}` znajdują się ścieżki dla odpowiedniego rozszerzenia.
6. Instrukcja `!COPY ** $(DIR)` kopiuje zbiory zależne do kartoteki zdefiniowanej przez makro `DIR`.

Opisy i przykłady zastosowań wielu innych komend i makr dla programu *Nmake* znajdują się w pracy [MC], natomiast dla programu *Make* w pracy [BC1].

7.3. Tworzenie bibliotek

Biblioteka jest plikiem utworzonym przez połączenie w jeden wielu plików typu `.OBJ`. Zmniejszenie liczby otwieranych plików pozwala na przyspieszenie pracy programu łączącego. Różnica między plikiem `.LIB` a modulem `.OBJ` uwidacznia się także w sposobie łączenia. Moduł `.OBJ` włączany jest zawsze w całości do kodu wynikowego, z pliku `.LIB` natomiast wybierane są tylko moduły zawierające wykorzystywane funkcje. Pamiętać także należy, iż w czasie pracy krokowej nie można prześledzić wykonania funkcji z biblioteki. Z tego względu należy tam umieszczać tylko wcześniej dokładnie przetestowane funkcje. Możliwość tworzenia własnych bibliotek jest zatem bardzo pomocna w oprogramowywaniu dużych systemów.

Najczęściej wykonywanymi operacjami na bibliotekach są: dodanie, usunięcie i wybranie modułu z biblioteki. Podczas wywołania programu obsługującego bibliotekę (*Tlib* dla Borlanda lub *Lib* dla Microsofta) w linii zlecenia jest podawana nazwa biblioteki i lista modułów, z których każdy poprzedzony jest kodem operacji, który może przyjmować następujące wartości:

- '+' dodanie modułu do biblioteki
- '-' usunięcie wyspecyfikowanego modułu
- '*' tworzenie zbioru `.OBJ` ze wskazanego modułu

Dopuszcza się też stosowanie kombinacji powyższych kodów, np. `'-+'` czy `'*-'`. Niemożliwe jest naturalnie dokonywanie operacji na pojedynczych funkcjach z modułu, moduł jest najmniejszą jednostką przetwarzania dla programu *Lib*.

W trakcie wykonywania operacji w bibliotece zalecane jest tworzenie protokołu, zawierającego nazwy funkcji znajdujących się w kolejnych modułach. Otrzymamy go podając po nazwie biblioteki i wykonywanych operacjach przecinek oraz nazwę pliku, do którego zostanie skierowany protokół.

7.4. Wyszukiwanie wzorców

Program **Grep** jest niczym innym jak rozszerzoną wersją systemowej komendy MS DOS **FIND**. Program **Grep** przegląda pliki w poszukiwaniu zadanego wzorca. Jeśli tylko zostanie znaleziony taki plik, to na urządzeniu wyjściowym otrzymamy nazwę pliku i wiersz zawierający ten wzorcowy ciąg. Rozszerzeniem w stosunku do komendy **FIND** jest zastosowanie wyrażeń regularnych do opisu wyszukiwanych wyrażeń.

Program ten jest bardzo przydatny do określenia miejsc definiowania i wykorzystania zmiennych, stałych i funkcji.

Poniżej przedstawiono najprostsza postać wywołania programu **GREP**,

```
grep Borland *.txt
```

po której jako wynik otrzymamy listę wierszy z plików zawierających ciąg "Borland". Ponieważ program domyślnie rozróżnia małe i duże litery więc, ciągi "borland" i "Borland" zostaną potraktowane inaczej.

Ogólnie postać syntaktyczna jest następująca:

```
grep [opcje] szukany_ciąg [plik(i)...]
```

Opcje to jedna lub wiele liter poprzedzonych symbolem minus (-). Każda indywidualna litera jest przełącznikiem. Symbol plus (+), występujący po literze, włącza ją i jest opcjonalny (-r oznacza to samo co -r+). Natomiast symbol minus (-) wyłącza. Listę opcji można kumulować (-ild) lub pisać rozdzielnie (-i -l -d). Jedynie opcja -r jest domyślnie włączona, wszystkie pozostałe są wyłączone. Każde powtórne wystąpienie opcji przesłania poprzednią definicję.

Znaczenie opcji:

- c Wyświetl, zamiast pełnych wierszy z wyszukanym ciągiem, tylko liczbę tych wierszy.
- d Przeglądaj także wszystkie podkartoteki.
- i Ignoruj wielkość liter, litery od a do z są wówczas identyczne z literami od A do Z.
- l Wypisz tylko nazwę znalezionej pliku.
- x Dopisz do każdego wiersza jego numer w pliku.

- o Wypisz używając unix-owego formatu wydruku.
- r Stosuj wyrażenie regularne. Ciąg do wyszukiwania nie jest przetwarzany literalnie (bezpośrednio znak po znaku). Następujące znaki są traktowane specjalnie i oznaczają:
 - ^ początek wiersza
 - \$ koniec wiersza
 - . dowolny znak
 - \ następny znak przetwarzany literalnie
 - * poprzedni znak zero lub więcej razy
 - + poprzedni znak jeden lub więcej razy
 - [aeiou0-9] skojarz a, e, i, o, u i od 0 do 9
 - [^aeiou0-9] skojarz cokolwiek oprócz a, e, i, o, u i od 0 do 9.
- u Zapisz wszystkie podane opcje jako domyślne do pliku GREP.COM.
- v Drukuj wszystkie nieskojarzone wiersze.
- w Szukaj ciągu zakładając, że jest on słowem, tzn. jest on ograniczony znakami różnymi od liter. Znakami tworzącymi słowa są litery od A do Z, cyfry od 0 do 9 i znak podkreślenia. Można także zdefiniować ten zbiór podając w[zbiór].
- z Drukuj nazwy przeszukiwanych zbiorów, każdy wiersz jest poprzedzony numerem i podaj globalną liczbę skojarzonych wierszy dla danego pliku.

Program pozwala na używanie standardowego wejścia (filtry i strumienie). Jeśli wynik nie mieści się na ekranie, to można składować go w pliku zewnętrznym. Jeśli po nazwie programu (GREP) zostanie podany znak zapytania, wydrukowane zostaną najważniejsze informacje o argumentach dopuszczalnych w linii zlecenia. Poszukiwany ciąg można umieścić w cudzysłowie, dzięki czemu spacje i tabulacje mogą zostać umieszczone w jego wnętrzu. Całkowita długość wywołania GREP nie może przekroczyć 80 znaków. Poniższe przykłady ilustrują zastosowanie tego programu:

```
grep -r [^a-z]main\ *( *.c
```

Poszukiwane są ciągi main, ale bez przedrostków z małymi literami ([^a-z]), po których może wystąpić dowolna liczba spacji (\ *) i otwierający nawias okrągły. Poszukiwanie odbywa się w bieżącej kartotece, po plikach z rozszerzeniem .c:

```
grep -ri [a-c]:\\data\\.fil *.c *.asm
```

W plikach o rozszerzeniach .c i .asm są poszukiwane ciągi zawierające oznaczenie napędu dysków [a-c], dwukropek, kreskę ukośną i nazwę pliku data.fil. Duże i małe litery nie są rozróżniane.

```
grep -ri [^a-z]słowo[^a-z] *.doc lub grep -iw słowo *.doc
```

Jest to przykład na wyszukiwanie konkretnego słowa.

```
grep -w[=] = *.c
```

Przykład ten redefiniuje bieżący zbiór znaków definiujących słowo i jedynym znakiem słowa stał się znak przypisania. Takie wywołanie usuwa z procesu przeszukiwania podwójne wystąpienie znaku równości.

7.5. Profilowanie programu

W procesie wytwarzania profesjonalnego oprogramowania ważną rolę odgrywa profiler. Na pytanie, co to jest profiler i jak się nim efektywnie posługiwać, odpowie ten podrozdział.

7.5.1. Definicja profilowania

Profilery, znane też jako analizatory działania programu, są programowymi narzędziami, które pokazują m. in.:

- gdzie program 'traci najwięcej czasu'
- ile razy wykonywana jest dana linia programu
- ile razy dana funkcja jest wywoływana i przez jakie inne funkcje
- które pliki są przetwarzane i przez jaki okres.

Profilery analizują też wykorzystanie zasobów komputera, m.in.:

- procesora
- dysków
- klawiatury
- drukarki
- przerw i nakładek.

Analizowanie pracy urządzeń I/O i dostarczanie szczegółowych raportów statystycznych dla dowolnej części programu umożliwia zharmonizowanie jego działania z otoczeniem systemowym. Pokazanie przez profiler najważniejszych parametrów wykonywania programu, a więc czasów działania, liczników wywołań, wykorzystywanych przerw systemowych i liczbyostępów do plików, pomaga w optymalizacji kodu i przyspiesza działanie programu.

Jaka jest różnica między programem optymalizującym a profilerem? Optymalizator przyspiesza działanie zastępując w standardowy sposób czasochłonne instrukcje lub nawet całe sekwencje przez mniej czasochłonne. Profiler tego nie wykonuje, pomaga natomiast wykryć najmniej sprawną część kodu wskazując na algorytmy, które mogą zostać zmodyfikowane lub na nowo napisane przez programistę. Kilkuletnie doświadczenie pokazuje, że lepsze rezultaty osiągane są wraz z modyfikowaniem algorytmów i struktur danych niż z optymalizacją kodu. Oszczędności osiągane dzięki automatycznej optymalizacji kodu tylko w wyjątkowych wypadkach pozwolą na skrócenie czasu wykonania o więcej niż 5%.

Próba znalezienia tzw. zatorów, czyli jałowych miejsc w programie bez profilera, to jak szukanie błędów wykonania bez używania debuggera. Profiler zdecydowanie redukuje czas i wkładany wysiłek.

W dalszej części omawiany jest profiler, Turbo Profiler 1.1 (TPROF), dostarczany w pakiecie Borland C++ 2.0.

Profilowanie programu odbywa się w czterech krokach:

1. Zakładanie obszarów profilowanych. Obszarem takim jest fragment, o którym chcemy otrzymywać informację. Może to być pojedynczy wiersz, instrukcja iteracyjna lub cała funkcja.
2. Zbieranie danych podczas uruchomienia programu. Najczęściej są to czasy wykonania, liczniki wywołań itp.
3. Analiza zebranych danych.
4. Modyfikowanie programu i ponowna jego kompilacja z włączeniem informacji dla debuggera (opcja -v).

Po zmodyfikowaniu programu należy powtórzyć kroki od 1 do 3, żeby sprawdzić, czy dokonane zmiany polepszyły działanie programu. Nie jest to proces liniowy, podane kroki nie muszą być wykonywane zawsze w podanej kolejności. Może się ona zmieniać w zależności od potrzeb. W omawianym profilerze są one wykonywane interakcyjnie.

Po stworzeniu charakterystyki, zanalizowaniu wyników i zmodyfikowaniu programu następuje powtórne profilowanie. Schemat postępowania - jak widać - jest dość prosty. Jednakże kłopoty z interpretacją wyników powodują, że działania optymalizujące wykonuje się metodą prób i błędów.

Już na samym początku należy zadbać o dobre przygotowanie przebiegu testującego profilowanego programu. Zbiór przetwarzanych danych musi być reprezentatywny i na tyle duży, by uzyskane wyniki

były użyteczne. Oznacza to, że czas wykonania programu powinien wynieść minimum 5 s. Najlepiej też usunąć z analizy wszystkie zależne od klawiatury części programu.

7.5.2. Obszary profilowane

Obszar profilowany to miejsce w programie, w którym będą zbierane dane statystyczne. Po zdefiniowaniu obszaru na jego początku jest umieszczany znacznik. Obszarem może być pojedyncza linia, instrukcja iteracyjna lub sekwencja instrukcji. Znacznik takiego obszaru spełnia rolę wewnętrznej pułapki. Po jego napotkaniu wykonywane są różne operacje – zależne od aktualnej konfiguracji profilera.

Podczas zakładania obszarów należy określić poziom szczegółowości uzyskiwanych informacji. Dla małego programu, mającego około 100 linii kodu, stosuje się maksymalny poziom szczegółowości, co oznacza że każda linia programu jest obszarem profilowanym. Dla dużych, wielomodulowych programów wystarczają zazwyczaj dane o czasach wykonania każdej funkcji. Jeśli kod źródłowy przekracza 10,000 linii, to powinno się analizować aktywnie (patrz p. 7.5.3) tylko jeden moduł. Jeśli program działa krócej niż 5 s, to aby uzyskać poprawne wyniki, należy skorzystać z możliwości samoczynnego wielokrotnego uruchomienia programu. Jeżeli natomiast program działa ponad godzinę, to trzeba zrezygnować z wielu obszarów profilowania (co przyspieszy jego działanie) i stosować tryb pasywny analizy (patrz p. 7.5.3). Dodatkowo dla dużych programów stosuje się filtrowanie otrzymywanych wyników.

Przed samym procesem profilowania należy tak zaznaczyć interesujące obszary, by zbierane statystyki były przydatne i wyczerpujące. W pierwszym przebiegu można skorzystać z domyślnie ustawionych znaczników dla obszarów profilowanych – Turbo Profiler ustawia je na podstawie tablicy symboli, znajdującej się w pliku wynikowym .EXE.

Dla właściwego definiowania obszarów profilowania celowe jest zaznajomienie się ze sposobem działania Turbo Profilera. Po każdorazowym napotkaniu znacznika obszaru profiler wywołuje wewnętrzną funkcję, która aktualizuje stos wywołań funkcji. Z każdym obszarem są związane dwie zmienne: licznik wywołań i czas wykonania. Kiedy tylko program wchodzi do obszaru, licznik wywołań jest inkrementowany. Po każdym takcie zegara inna wewnętrzna funkcja inkrementuje zmienną czas, związaną z bieżącym obszarem profilowania.

Użytkownika może interesować całkowity czas wykonania funkcji wraz z funkcjami przez nią wywoływanymi, bądź też czas bez tych wywołań. W celu rozróżnienia tych dwóch sposobów pomiaru czasu funkcji, profiler stosuje dwa rodzaje znaczników: włączający (opcja *Combined*) i wyłączający (opcja *Separate* - domyślny).

Po napotkaniu znacznika wyłączającego nowy znacznik staje się znacznikiem bieżącym, na wewnętrzny stos wywołań funkcji jest odkładana funkcja wywołująca wraz z jej parametrami. Po napotkaniu instrukcji *return* ze stosu jest pobierana poprzednia wartość parametrów i odpowiadający im obszar staje się bieżący.

W przypadku znacznika włączającego działania te nie są wykonywane i program profiler pomija wszystkie zdefiniowane obszary, aż do wyjścia z tej funkcji. Definicję znacznika włączającego powinno się stosować tylko przy wejściu do funkcji.

Turbo Profiler może zapamiętać historię każdego wywoływania zaznaczonej funkcji. Po napotkaniu znacznika funkcja staje się bieżąca i są tworzone nowe zmienne: czasu i licznika wywołań, a poprzednie wartości są zapamiętywane na stosie wywołań. Po kolejnym wywołaniu tej samej funkcji następuje zapamiętanie bieżącej historii wywołań, jeżeli na stosie wywołań nie została ona już zapamiętana.

Ważnym parametrem może być sam czas wywołania funkcji (tj. czas operacji na stosie koniecznych do uruchomienia i dezaktywacji funkcji), a nie całkowity czas jej działania. Automatycznie wykonywane jest to podczas analizy pasywnej (patrz p. 7.5.3). Innym, łatwym sposobem obliczenia czasu wywołania jest zaniechanie zbierania informacji na samym początku funkcji i odblokowanie tego przy instrukcji powrotu. Można to osiągnąć ustawiając znacznik na pierwszym wierszu tej funkcji (opcja *Disable*) i na ostatnim (opcja *Enable*).

7.5.3. Monitorowanie programu - analiza aktywna i pasywna

Zwykle programiści używają profilera, aby otrzymać odpowiedzi na następujące pytania:

1. Jak wydajny jest algorytm?
2. Czy program wykonuje dokładnie to, co było zaplanowane?
3. Jak długo wykonują się poszczególne funkcje?
4. Czy struktura kodu jest odpowiednia?

Do otrzymania wiarygodnych odpowiedzi konieczne jest właściwe profilowanie programu. Proponujemy podporządkowanie się następującym regułom.

Podczas analizy algorytmu należy upewnić się, czy zaznaczono wszystkie wiersze i funkcje algorytmu. W przypadku dużego systemu nie profilujemy całego programu, ale wybieramy interesujący nas moduł. By pozbyć się zbędnej informacji, zakładamy filtr (opcja **Filtr**) lub usuwamy niechciane znaczniki obszarów (opcja **Remove**). Gdy testujemy nieznaną program, najpierw analizujemy drzewo wywołań (okno **Callers**), potem poszczególne czasy wykonania, a na końcu liczniki wywołań. W ten sposób znajdujemy najważniejsze funkcje i przewidujemy skutki, jakie mogą powstać po dokonaniu zmian w źródle. Dużo prościej utrzymujemy też kontrolę nad urządzeniami wejścia/wyjścia, plikami i przerwaniach.

Przed przystąpieniem do monitorowania działania programu należy dokonać wyboru sposobu zbierania informacji podczas profilowania. Mamy do dyspozycji dwie różne metody. Pierwsza z nich to analiza aktywna (domyślnie ustawiona). W jej trakcie są zbierane automatycznie m.in. czasy wykonania, ustawiane liczniki wywołań, jest tworzona historia wywołań funkcji, czy nawet są śledzone przerwy systemu DOS.

Jeśli program działa wolno i nie interesuje nas pełna analiza wywołań funkcji, to należy używać analizy pasywnej. Podczas takiego działania profiler zbiera tylko czasowo zależne statystyki dla zaznaczonych obszarów (czasy wykonania, śledzenie przerw i operacje na plikach). W ten sposób program wykonuje się znacznie szybciej niż w czasie analizy aktywnej. W analizie pasywnej profiler przerywa działanie programu w regularnych odstępach czasu i sprawdza rejestr CS:IP. Jeśli licznik ten wskazuje na zaznaczony obszar, to inkrementuje się odpowiednia zmienna czasu. Może więc się zdarzyć, że niektóre z zaznaczonych obszarów nie zostaną objęte analizą. Jeżeli program jest wykonywany krótko (np. kilkadziesiąt sekund), to aby zwiększyć wiarygodność uzyskanych danych statystycznych, zaleca się ustawienie opcji samoczynnego wielokrotnego uruchamiania.

Czasami profiler może działać zbyt wolno w stosunku do rzeczywistego czasu działania programu. Może to być spowodowane częstym wywoływaniem głęboko zagnieżdżonych funkcji, gdy jest ustawione śledzenie stosu dla wszystkich wywołań we wszystkich obszarach. Jeśli tylko taka funkcja została zdefiniowana jako obszar badany, to profiler poświęca na jego obsługę wiele czasu. Aby określić, czy rzeczywiście dużo czasu traci się bezpowrotnie w ten sposób, należy w oknie wykonywania sprawdzić obszary profilowania, posortowane według liczby wywołań. Jeśli program wywołuje wielokrotnie jakąś funkcję częściej niż pozostałe, to należy założyć filtr bieżący na tę funkcję lub też usunąć związany z nią obszar.

7.5.4. Interpretacja wyników i poprawianie programu

Zajmijmy się teraz analizą wyników, które znajdują się w oknach źródeł i raportów. Do pierwszej grupy zaliczamy okna: **Modul** (Moduł), **Area** (Obszar), **Function** (Funkcja) i **Processor** (Procesor). Na tym etapie są one używane do analizy porównawczej źródeł i otrzymanych raportów. Do drugiej grupy zaliczamy okna: **Execute** (Uruchomienie), **Caller** (Wywołanie), **Overlay** (Nakładka), **Interrupt** (Przerwanie) i **File** (Plik). Znajdują się tam dane statystyczne zebrane podczas uruchamiania programu. Okno **Execute** pokazuje posortowane obszary według jednego z kilku parametrów, np. czasu wykonania. Okno **Caller** zawiera listy wszystkich obszarów, które wywoływały wybraną funkcję oraz liczbę tych wywołań. Okno **Overlay** pozwala wykryć zbyt często używane nakładki, które powinny znaleźć się w module nienakładkowym. Okno **Interrupt** informuje o przerwaniach używanych przez program. Ostatnie okno **File** obrazuje liczbę operacji czytania i zapisu, dokonanych na plikach, wskazuje przy tym, który z plików jest najczęściej czytany lub modyfikowany.

Poniżej przedstawiono wiele zaleceń pozwalających na identyfikowanie funkcji, których prosta modyfikacja może znacznie poprawić efektywność działania programu.

1. Szukanie dużych funkcji z nieproporcjonalnym podziałem wewnętrznym czasu wykonania lub funkcji wielokrotnie wywoływanych.
2. Szukanie instrukcji i funkcji z największym stosunkiem czasu do liczby wywołań. Przedstawiając obrazowo, oznacza to szukanie obszarów z długim słupkiem czasu i krótkim słupkiem liczby wywołań.
3. Optymalizacja najbardziej zagnieżdżonych pętli przez zapamiętywanie tymczasowych wyników liczonych w każdej iteracji lub zastosowanie języka *Assembler*. Przeniesienie możliwych operacji poza pętlę czy też rozpisywanie krótkich pętli na bloki instrukcji.
4. Modyfikowanie struktur danych i algorytmów, np. algorytm *QuickSort* generalnie jest lepszy od sortowania bąbelkowego dla losowego rozkładu kluczy, unikanie operacji zmiennoprzecinkowych i stosowanie wskaźników daje zwykle duży zysk na czasie wykonania.
5. Zastosowanie metody *cache* dla najczęstszych operacji dyskowych. Funkcja *getc* czyta cały sektor z dysku do bufora, jej wynikiem jest jednak tylko pierwszy bajt. Następne wywołanie *getc* wykorzystuje już informację zawartą w buforze, aż do jego wyczerpania. We wszelkiego rodzaju edytorach można stosować bufor, które są aktualizowane podczas czekania na akcję użytkownika, np. bufor

strony pierwszej i ostatniej, bufor strony następnej i poprzedniej. Dzięki temu najczęściej wykonywane operacje mogą być wykonane bez konieczności czytania z dysku.

6. Zachowanie odpowiedniej kolejności podczas obliczania wyrażeń logicznych, tak by jak najszybciej określić wartość wyrażenia: w sekwencji operatorów AND na początku umieszczać wyrażenia najrzadziej zachodzące, a w sekwencji OR najbardziej prawdopodobne. Celowe jest też stosowanie różnego rodzaju zmiennych pomocniczych z częściowymi wynikami obliczeń.
7. Optymalizacja funkcji i wyrażeń. Przepisanie często wywoływanych funkcji na postać typu inline. Dopisanie kofunkcji dla operacji, np. `setjmp` i `longjmp` (patrz [BC4]). Używanie funkcji dających w wynikach złożone struktury i zamiast tablicowego indeksowania przez operacje na wskaźnikach (patrz p. 2.3.3).

7.5.5. TFREMOTE, TF386 i TFINST

Na koniec warto wspomnieć o kilku innych możliwościach Turbo Profilera i jego otoczenia. Podobnie jak w Turbo Debuggerze umożliwiono zdalne profilowanie programów (program **TFREMOTE**). Posiadacze procesora 80386 lub 80486 i pamięci rozszerzonej o pojemności co najmniej 700 KB mogą w pełni korzystać z wirtualnego profilowania (program **TF386**). Natomiast program **TFINST** pomaga dostosować profiler do wymagań użytkownika.

Jeśli program wymaga dużo pamięci, to może się zdarzyć, że TPROF nie pomieści się z profilowanym programem. W tej sytuacji otrzymamy komunikat "Not enough memory...". Narzędzie TFREMOTE rozwiązuje ten problem pozwalając uruchomić TPROF na jednym systemie, zaś sam program na innym systemie, połączonym przez port szeregowy z tym pierwszym. Instalacja przebiega następująco:

1. Skopiuj TFREMOTE i wszystkie pliki wymagane przez profilowany program na zdalny system. Można się posłużyć programem TDRF z otoczenia narzędziowego Turbo Debuggera 2.5. Profilowany program nie musi być skopiowany.
2. Podczas uruchamiania zdalnego należy w linii zlecenia podać najpierw komendę **TFREMOTE -rpl(2) -rs4(3,2,1)**, której pierwszym parametrem jest numer portu COM1 lub COM2, natomiast drugi parametr odpowiada za szybkość transmisji, od 4 (115000 bodów) do 1 (9600 bodów). Następnie z tymi samymi parametrami wywołać TPROF.

3. Po uruchomieniu TFREMOTE i TPROF w trybie zdalnym proces profilowania przebiega tak samo jak na pojedynczym komputerze. W razie nieprawidłowego działania, najpierw należy sprawdzić połączenie między komputerami, a następnie spróbować uruchomić program z mniejszą prędkością transmisji, np. -rs1.

Lepszym rozwiązaniem kłopotów z pamięcią jest wirtualne profilowanie, które pozwala korzystać z pełnej przestrzeni adresowej 640 KB, ponieważ profiler jest ładowany powyżej 1 MB. Z punktu widzenia użytkownika nie następuje żadna zmiana. Program wykonuje się jak w normalnym trybie, ale z minimalną utratą szybkości wykonania. Jedynymi warunkami pracy wirtualnej jest posiadanie procesora 80386(486) i około 700 KB wolnej pamięci rozszerzonej. Przed uruchomieniem TF386 należy upewnić się, czy został zainstalowany driver TDH386.SYS. W przypadku większego niż 256 bajtów obszaru na zmienne systemowe, instalacja drivera powinna mieć np. postać DEVICE = TDH386.SYS -e800, dzięki czemu rezerwujemy 800 bajtów na te zmienne.

Podobnie, jak i dla innych produktów Borlanda, można dokonywać zmian wartości domyślnych Turbo Profilera, korzystając z programu TFINST. Są to:

- kolory i wzorce dla okien ekranowych
- formaty wyświetlania, tryb przełączania ekranu
- wielkość ekranu, aktualizacja ekranu użytkownika
- komendy edytora i nazwy kartotek podczas wyszukiwania
- opcje językowe, mysz, długość ścieżek wywołań
- specyfikacja pamięci rozszerzonej, opcje zdalnego profilowania.

7.6. Usuwanie błędów w programach

Częstym błędem, popełnianym szczególnie przez początkujących programistów, jest przystępowanie do pisania programu przed jego starannym zaprojektowaniem. Jest powszechnie znane, że jedna godzina pracy projektowania zaoszczędza wiele godzin żmudnego szukania błędów. Podczas kodowania nawet najlepiej zaprojektowanego programu nieuchronne jest jednak popełnianie błędów. Dlatego bardzo dobry programista musi być zarazem dobrym 'znajdowaczem' błędów. W języku C występują standardowe, jak i 'dość' specyficzne błędy. Celem tego rozdziału jest zwrócenie na nie uwagi. Poznając typowe błędy poprawiamy naszą umiejętność pisania bezbłędnych programów.

7.6.1. Kolejność wykonywania operacji

Błąd ten najczęściej jest spotykany podczas poprawiania istniejącej już instrukcji programowej, np. instrukcja

```
x = *p++;
```

przypisuje zmiennej *x* wartość wskazywaną przez wskaźnik *p* i następnie inkrementuje ten wskaźnik. Załóżmy, że należy zmienić tę instrukcję i zmiennej *x* przypisać wartość wskazywaną przez *p*, pomnożoną przez tę samą wartość. Początkujący programista na ogół zapisze tę instrukcję następująco:

```
x = *p++ * *p;
```

Jednakże nie jest to prawidłowe rozwiązanie tego problemu. Prawidłowo wygląda to tak:

```
x = *p * *p++;
```

Błędy tego typu jest bardzo trudno znaleźć i jedynym skutecznym sposobem jest dokładne przeanalizowanie kolejności operacji w danym wyrażeniu. Tabelę priorytetów i sposobu łączenia operatorów zamieszczono w załączniku (p. 8.5).

Wszystkie operatory podzielono na 15 grup według malejącego priorytetu. Na poziomie grupy priorytet i sposób łączenia są te same. Łączenie lewe (prawe) oznacza, że dany operator wiąże operand lub operandy ze strony lewej (prawej) na prawą (lewą) stronę.

Jeżeli co najmniej dwa operatory z jednej grupy dotyczą tego samego argumentu, to kolejność wykonywania jest określona przez łączenie. Rozważmy wyrażenie: $x*=-i++y-z$!

1. $-i++$

Na argument *i* działają dwa operatory o równym priorytecie: unarny minus i postinkrementacja. Ponieważ łączenie następuje ze strony prawej na lewą, to wyrażenie to można zastąpić $-(i++)$.

2. $x*=(-(i++))+y-z$

Ponieważ priorytety '+' i '-' są wyższe od priorytetu '=' i łączy argumenty z lewej strony, więc wyrażenie przyjmuje postać: $x*=(((-(i++))+y)-z)$.

7.6.2. Problemy ze wskaźnikami

W przypadku zmiennych wskaźnikowych należy zapamiętać, że pierwszą czynnością po zdefiniowaniu wskaźnika jest nadanie mu wartości, tzn. przydzielenie mu pamięci. Rozważmy fragment programu:

```
int *x;
*x = 100;
```

Takie użycie może oznaczać szybką katastrofę dla programu. Nie wiadomo przecież, gdzie wskazuje *x*, mogą to być bajty któregoś przerwania lub adres powrotu z funkcji. Wpisanie pod tym adresem wartości 100 może spowodować zablokowanie systemu.

Opisywana sytuacja jest wykryta przez kompilator po włączeniu najwyższego poziomu wykrywania błędów (kompilator Borland C++ 2.0 opcja `-w`, co jest najlepszym sposobem pomocy programiście podczas wyszukiwania błędów) i zostanie wypisany komunikat `'possible use of 'x' before definition in function..'`. Dzieje się jednak tak tylko wtedy, gdy *x* jest zmienną lokalną. Gdy jest to zmienna globalna, ostrzeżenie takie nie pojawi się. W zamian za to wystąpią błędy wykonania, a po zakończeniu może pojawić się komunikat `'Null pointer assignment'`. Należy pamiętać jednak, że komunikat ten nie zostanie wyprowadzony przez programy skompilowane w modelu TINY, LARGE i HUGE.

Komunikat ten sygnalizuje wpisanie wartości pod adresem zerowym, co jest ewidentnym błędem. Adres zerowy oznacza tutaj wskaźnik zdefiniowany jako NULL. Jako przykład takiego błędnego programu niech posłuży przykład:

```
char *wsk=NULL;
strcpy(wsk,"SINT");
```

Stała NULL jest wykorzystywana powszechnie do sygnalizowania niepoprawnego wykonania funkcji operującej na wskaźnikach. Komunikat ten może zostać wyświetlony, bowiem po zakończeniu wykonywania każdego programu skompilowanego w odpowiednim modelu (patrz wyżej), w Borland C++ 2.0 jest wykonywana następująca sekwencja:

```
xor ax,ax
mov si,ax
mov cx,002F
et: add al,[si]
    adc ah,00
    inc si
    loop et
    sub ax,0D5C
```

Jak widać, jest sprawdzanych 47 bajtów rozpoczynających segment. Odbywa się to przez ich sumowanie. Otrzymanie w wyniku wartości różnej od 0xd5c powoduje wypisanie komunikatu.. Analogiczna sekwencja instrukcji w programach skompilowanych za pomocą Microsoft C 6.0 wygląda następująco:

```
push si
xor si,si
mov cx,0042
xor ah,ah
cld
et: lodsb
xor ah,al
loop et
xor ah,55
```

Sprawdzanych jest 66 początkowych bajtów segmentu danych. W miejsce sumowania jest wykonywana różnica symetryczna kolejnych bajtów i ostateczny wynik jest ponownie różnicowany symetrycznie z wartością 0x55. W rezultacie, jeżeli którykolwiek bajt uległ zmianie, jest uruchamiana funkcja wypisująca komunikat o błędzie.

W obu przypadkach testowany obszar pamięci zawiera między innymi nazwę kompilatora i komunikat 'Null pointer assignment'.

7.6.3. Brak deklaracji funkcji i błędne podanie argumentów

Podstawowa zasada programowania jest niewprowadzanie własnych funkcji o tych samych nazwach co funkcje standardowe. Początkujący programiści, gdy wreszcie zrozumieją konsekwencje tego, że printf w języku C jest funkcją, mogą poczuć pokusę przeddefiniowania standardowych funkcji. Konsekwencje tego kroku mogą być bardzo duże. Najgorszą sytuację mamy wtedy, gdy standardowa funkcja zostanie przeddefiniowana i właśnie ona jest używana przez inną standardową funkcję. Taki przypadek może się zdarzyć podczas redefinicji funkcji *getc* i używania w programie funkcji standardowej *scanf*. Taki błąd jest prawie nie do wykrycia. Jedyną bronią jest np. rozszerzanie nazw tworzonych funkcji o jakiś niespotykany przedrostek.

Za każdym razem, gdy tylko wynikiem funkcji jest inny typ niż *int* i jej definicja znajduje się po jej wywołaniu, w bibliotece lub w innym pliku, musi nastąpić jej deklaracja (definicja prototypu funkcji). W przeciwnym przypadku kompilator potraktuje tę funkcję jako typu *int*, co może doprowadzić do błędnego działania programu.

Klasyczne przykłady błędnego działania programu z tego powodu występują przy funkcjach o wartościach typu float lub wskaźnik. Gdy brakuje deklaracji (w sytuacji opisanej jak powyżej) podczas przypisania zmiennej typu float wartości takiej funkcji, dokonuje się rzutowanie na typ całkowity i 32 bity informacji są redukowane do 16 (wielkość zmiennej typu int). Podobnie dzieje się z funkcją typu wskaźnikowego. Następuje niejawną konwersja na typ int i otrzymujemy błędny wynik, co może mieć bardzo znaczące konsekwencje.

Podobna sytuacja występuje w przypadku argumentów funkcji. Najlepszym przykładem jest funkcja scanf. Jak wiadomo, argumentami scanf muszą być wskaźniki i dlatego zapis

```
int x;  
char string[10];  
scanf("%d%s", x, string);
```

jest błędny i prawidłowe wywołanie powinno mieć postać:

```
scanf("%d%s", &x, string);
```

Dla tablic nie musimy używać operatora &, gdyż wartością wyrażenia, jaką jest nazwa tablicy, jest wskaźnik.

7.6.4. Przepelnienie stosu

Podczas wywoływania funkcji na stos odkładane są: adres powrotu do wywołującej funkcji, argumenty i dynamicznie alokowane zmienne lokalne. Stos znajduje się między kodem a stertą. Jego rozmiar jest z góry określony i w trakcie działania programu nie ulega zmianie. Jeśli stos zostaje przepełniony, to wchodzi on w obszar sterty. Niestety, zwykle nie pojawia się wtedy żadne ostrzeżenie, a program zawiesza się. Sytuacji tej unikniemy przez zastosowanie opcji kompilatora -N (Borland) lub /Ge (Microsoft). Spowoduje ona dołączenie kodu sprawdzającego przepełnienie stosu. Wydłuża to jednak czas działania programu.

Jeśli pojawi się komunikat 'Stack overflow', to należy zwiększyć obszar przeznaczony na stos. Gdy zależy nam na oszczędnym gospodarowaniu pamięcią, to możliwe jest jego zmniejszenie. W kompilatorze Borland C++ 2.0 osiąga się to definiując zmienną globalną _stklen i inicjując ją wielkością stosu podaną w bajtach:

```
unsigned _stklen = 20000; /* wielkość stosu 20 000 bajtów */
```

Domyślną wartością są 4 KB. Kompilator Microsofta nie pozwala na programową definicję stosu. Można natomiast dokonać tego na trzy inne sposoby, które mają wpływ tylko na kod wynikowy:

1. Ustawiając opcję kompilatora /F, np. /F 4A00. Wartość występująca po spacji jest liczba szesnastkowa, definiująca wielkość stosu w bajtach. Domyślnie stos jest ustawiony na 2 KB.
2. Stosując opcję linkera /ST[ACK]:liczba.
3. Używając programu narzędziowego EXEHDR z parametrem /STACK:liczba.

Wydaje się, że teraz wszelkie kłopoty są już rozwiązane. Nie sposób jednak pokonać możliwości sprzętowych. We wszystkich przypadkach największą wartość, jaka może przyjąć rozmiar stosu, jest długość segmentu, czyli 64 KB. Jest to wystarczające w większości przypadków i dopiero w wyniku używania rekurencji może dojść do przepełnienia się stosu. W takim przypadku jedynym wyjściem jest przekształcenie rekurencji w iterację. Czasami też algorytm rekurencyjny jest gorszy od iteracyjnego. Rozważmy obliczenie kolejnych liczb Fibonacciego według schematu:

1. $f(n+1)=f(n)+f(n-1)$ dla $n>1$
2. $f(0)=0$, $f(1)=1$

Narzucającym się sposobem implementacji algorytmu jest zdefiniowanie funkcji rekurencyjnej f :

```
unsigned long f(int arg)
{
    if (arg==0)
        return 0L;
    if (arg==1)
        return 1L;
    return (f(arg-1) + f(arg-2));
}
```

Zauważmy, że dla $n>1$ każde wywołanie funkcji f powoduje dalsze dwa wywołania, i co – za tym idzie – kolejne adresy powrotu i argumenty są odkładane na stosie. Prowadzi to do szybkiego przepełnienia i wolnego działania samego algorytmu. Wyjściem jest algorytm iteracyjny, dużo szybszy od poprzedniego i standardowo korzystający ze stosu:

```
unsigned long f(int arg)
{
    unsigned long i=1L, fi=1L, y=0L, tmp;
    if (arg==0)
        return 0L;
    while (i<arg)
    {
        tmp=fi, i++;
        fi=y;
    }
```

```
        y=tmp;
    }
    return fi;
}
```

Zalecamy uruchomienie obu wersji funkcji *f* i porównanie szybkości ich działania.

7.6.5. Błędy spotykane podczas programowania w języku C

Oprócz błędów opisanych wcześniej szczegółowo, wymienić należy jeszcze kilka innych godnych uwagi. Oto one:

1. Efekt uboczny

Jeśli nieuważnie używa się zmiennych globalnych w funkcjach, to wywołanie takiej funkcji może pozostawić po sobie nieoczekiwany rezultat:

```
char tablica[20];
strcpy(tablica,"Dobrze\n");
iteracja("NIC");
puts(tablica);
...
iteracja(char *wsk)
{
    strcpy(tablica, wsk);
    while(*wsk)
        ...
}
```

Poprawnym działaniem jest używanie zamiast tej samej zmiennej globalnej lokalnego bufora dla funkcji *iteracja*.

Inny błąd tego typu może powstać w wyniku definiowania zmiennych zewnętrznych o często stosowanych, popularnych nazwach. Rozważmy poniższy fragment kodu:

```
int i;
f()
{
    int j;
    for (i=0; i<100; ++i)
    {
        ...
    }
}
```

```

    g();
    }
}

g()
{
    ...
    for (i=0; i<90; ++i)
    ...
}

```

Zmiana liczby wykonań instrukcji iteracyjnej `for` w wyniku wywołania funkcji `g` jest najprawdopodobniej niezamierzona. W rzeczywistym programie fakt, że `i` jest zmienną zewnętrzną jest z reguły znacznie mniej widoczny. Wynika stąd, że zmienne zewnętrzne powinny mieć długie, znaczące nazwy.

2. Brak dealokacji zmiennych

Błąd ten jest szczególnie groźny, gdy nie zostanie zwrócona pamięć przydzielona wewnętrznemu wskaźnikowi, tak jak funkcji `zla`:

```

int zla()
{
    char *robocza;
    robocza=malloc(1000);
    return(1);
}

```

Po powrocie z funkcji `zla` program nie ma już możliwości odzyskania pamięci w niej zaalokowanej. W krótkich przebiegach testowych błąd ten może pozostać niezauważony, lecz gdy program pracuje normalnie i po wielokrotnym wywołaniu funkcji `zla`, takie postępowanie prowadzi do całkowitego wykorzystania sterty. W trakcie uruchamiania programu jest zalecane kontrolowanie wartości funkcji `coreleft` (patrz [BC4]). Szczególnie przydatne są do tego celu debuggery, pozwalające na wpisywanie wartości funkcji jako wartości kontrolnej (patrz p. 7.6.6).

3. Zapominanie o ostatnim przejściu w pętli

```

for ( i=1; i<10; i++)
{
    ...
}

```

Instrukcja ta spowoduje tylko dziewięć wykonań pętli `for` zamiast dziesięciu planowanych.

4. Używanie niezainicjowanych zmiennych automatycznych


```
funkcja(void)
```

```
{
    int i;
    while (i++<10)
        ...
}
```

Zmienna *i* nie jest zdefiniowana i instrukcja iteracyjna *while* wykonuje się nieskończoną liczbę razy.

5. Błędy spowodowane podobieństwem operatorów `==` i `=`

Instrukcja `if ( x = y )` spowoduje podstawienie pod *x* wartości *y*, po czym nastąpi sprawdzenie, czy *y* jest różny od zera. Zapewne powinno być `if ( x == y )`.

6. Niewłaściwa arytmetyka na wskaźnikach

```
int *wsk;
wsk += sizeof(int);
```

Taka inkrementacja nie wskazuje na następny element tablicy *wsk*, ponieważ wskaźnik zwiększył się o cztery bajty zamiast o dwa. W języku C automatycznie jest uwzględniana wielkość wskazywanego elementu. Poprawna instrukcja to *wsk++*.

7. Niespodziewana propagacja bitu znaku

```
int i = 0xFFFE;
long l = i;
if ( l & 0x80000000L)
```

Wartość `l & 0x80000000` jest różna od 0, ponieważ po podstawianiu wartości skalarnych typu *int* pod typ *long* znak dodatni lub ujemny jest zachowany, np. wartość typu *int* `-2` (`0xFFFE`) po podstawieniu za liczbę typu *long* daje `-2L`, tzn. `0xFFFFFFF`.

Jeszcze bardziej nieoczekiwany wynik otrzymamy w wyniku wykonania poniższego kodu:

```
char i=200;
long l=99999L;
if (l<i)
    puts("i jest większe");
```

Propagacja znaku podczas obliczania wartości wyrażenia `l<i` spowoduje, że komunikat "i jest większe" zostanie wyprowadzony.

8. Nieoczekiwana redukcja bitu znaku

```
long l = 0x10000L;
int i = 1;
while ( i > 0 )
```

Warunek w tym przypadku nie jest spełniony, gdyż podstawienie wartości 1 pod i spowoduje obcięcie 16 bardziej znaczących bitów, pozostawiając jedynie zera.

9. Przetwarzanie średników

```
for (i=0; i<10; i++);
{
...
}
```

Kod ten wydaje się poprawny, ale kod w bloku po instrukcji for wykona się tylko jeden raz. Bliższa analiza wykaze nadmiarowy średnik po instrukcji for. Wykrycie tego błędu bez debuggera może trwać cały semestr.

10. Efekt uboczny w makrach

```
#define TOUPPER(c) 'a'<=(c) && (c)<='z' ? (c)-'a'+'A' : (c)
TOUPPER(*c++);
```

Wskaźnik c jest inkrementowany dwa lub nawet trzy razy i zależy to tylko od znaku: czy jest małą literą czy dużą.

11. Przesłanianie zmiennych

```
funkcja(void)
{
int i;
for (i=10; i>=0; i--)
{
int i;
if (i == 0)
...
}
...
}
```

Powtórne definiowanie zmiennej i powoduje, że jest nieokreślone czy warunek i==0 jest spełniony czy też nie. Pętla zostanie wykonana właściwą liczbę razy.

12. Adres zmiennej lokalnej wynikiem funkcji

```
int *funkcja(int j)
{
```

```
    int i=j/5;
    return(&i);
}
```

Jest to oczywisty błąd, gdyż zwracamy adres do nieistniejącej już zmiennej. Przypomnienie: zmienne lokalne dynamiczne są przechowywane na stosie, a nie w segmencie danych.

13. Brak instrukcji return

```
char *funkcja(char *wsk)
{
    while(*wsk)
    {
        if('A' <= *wsk && *wsk <='Z')
            return(wsk);
        wsk++;
    }
}
```

W przypadku braku w ciągu wsk dużych liter zostaną zwrócone wartości losowe. Na końcu powinno znaleźć się return(0).

14. Słowo kluczowe break

```
for (...)
    while (...)
        if (...)
            break;
...
```

Instrukcja break powoduje wyjście tylko z pętli while. Jest to jeden z nielicznych przypadków uzasadnionego użycia instrukcji goto, by wyjść także z drugiej pętli.

15. Brak generowania kodu

Ostrzeżenie "Code has no effect" otrzymamy po napisaniu instrukcji: a+b; – najprawdopodobniej programista chciał napisać a+=b lub inną podobną instrukcję.

7.6.6. Stosowanie debuggera

W ostatnich latach nastąpił ogromny rozwój debuggerów. Wiele w tej dziedzinie zawdzięczamy firmie Microsoft, która jako pierwsza wypuściła na rynek debugger symboliczny CodeView. W ślad za nią podążyła firma Borland i pojawił się Turbo Debugger. Oba te programy przeszły długą drogę rozwoju i w obecnym kształcie są wzorcowymi programami w swojej grupie.

Podstawowymi krokami pracy z debuggerem są:

1. Zaprojektowanie ekranu (wybranie trybu przeglądania zbioru źródłowego, ustawienie okna dla pamięci, rejestrów i zmiennych itp.).
2. Ustawienie pułapek programowych (pułapki na instrukcje, na zmianę bajtów pamięci, na spełnienie się wyrażenia itp.).
3. Uruchomienie programu, obserwowanie zmian na ekranie. Program można wykonywać krokowo (wchodząc do własnych funkcji lub nie), zatrzymując się na ustawionych pułapkach.

Zakładamy, że podstawowe operacje, takie jak: podglądanie zawartości zmiennych i pamięci, praca krokowa, ustawianie pułapek itp. są znane i docenia się ich ogromną przydatność. Wydaje się, że stosowanie debuggera w języku C jest ważniejsze niż w innych językach z racji nagminnego używania wskaźników i braku testów poprawności na etapie kompilacji, np. wyrażen indeksujących w tablicach.

Ponieważ szczegółową instrukcję obsługi można przeczytać w dokumentacji [BC5] i [MC], nie zamieszczamy tutaj opisu wszystkich dostępnych operacji. Scharakteryzowane zostaną jedynie ciekawsze rozwiązania podczas poszukiwania błędu.

7.6.6.1. CodeView 3.0

1. Stosowanie wielu okien

W tym samym czasie można otworzyć dwa okna dla programu źródłowego. Okna te mogą pokazywać dwa różne fragmenty tego samego programu lub dwie postacie źródłowe tego kodu, np. w języku C i odpowiadającą mu postać assemblerową. W drugim przypadku zostaje zachowana równoległość śledzenia programu. Przełączanie między oknami jest możliwe w dowolnym momencie pracy.

2. Wywoływanie funkcji

Dowolna funkcja w języku C, także znajdująca się w bibliotece, może zostać wywołana w oknie komend lub oknie podglądania zmiennych według następującego formatu: ?nazwa_funkcji (lista_zmiennych). Po wykonaniu funkcji jej rezultat jest wyświetlany w oknie komend. Wywołanie takiej funkcji nie musi znajdować się explicite w programie. Wystarczy, że zostanie dołączony odpowiedni kod do listy pozostałych zbiorów .OBJ, wyspecyfikowanych w zbiorze dla linkera.

3. Sprawdzenie niezdefiniowanych wskaźników

Dostęp do danych przez niepoprawnie zdefiniowane lub niezainicjowane wskaźniki może prowadzić do nieprzewidzianego zachowania się komputera. System operacyjny w niektórych przypadkach ostrzega o tym (patrz p. 7.6.2). CodeView może zlokalizować ten błąd dzięki temu, że dysponuje historią wywołań i można ustalić ostatnią linię programu, wykonaną bezpośrednio przed ostrzeżeniem. W przypadku systemu DOS globalne i statyczne zmienne są inicjowane i przypisywana jest im wartość 0. Założenie warunkowej pułapki, testującej zmianę bajtu zerowego, spowoduje zatrzymanie wykonania i umożliwi poprawienie błędu. Bajt zerowy dla bliskich wskazań oznacza DS:0000, a dla wskazań dalekich 0000:0000.

4. Efektywne korzystanie z pułapek

Pułapki znacznie spowalniają działanie programu. Mając do dyspozycji komputer, bazujący na procesorze 80386(486) i używając opcji /R w linii zlecenia, udostępniamy dodatkowo cztery rejestry, dzięki czemu założone pułapki mogą być analizowane sprzętowo. Często stosowaną czynnością podczas śledzenia jest ustawianie krotności pomijania danej pułapki. Program przerywa działanie dopiero po kilkakrotnym przejściu przez punkt kontrolny (opcja Pass Count).

5. Wydruk kontrolny

Komenda drukuj pozwala na wydrukowanie dowolnej części okna. Tekst zaznacza się za pomocą myszy lub naciskając jednocześnie Shift i dowolny klawisz kursora. Domyślnie tekst kopiowany jest do zbioru CODEVIEW.LST w bieżącej kartotece. Zmieniając nazwę tego pliku na LPT1 lub COM2 kierujemy wydruk bezpośrednio na drukarkę.

6. Zmienne rejestrowe

Zmienne te mogą powodować problemy podczas śledzenia programu. Nie zawsze podczas wyświetlania zawierają one bieżącą wartość. Generalnie należy wyłączać wszelką optymalizację kompilatora i deklarowanie zmiennych rejestrowych podczas procesu śledzenia.

7. Używanie dodatkowej pamięci

Wybierając opcję /X lub /E tablica symboli wraz z debuggerem jest ładowana do pamięci powyżej 1 MB. Opcja /X używa pamięci w standardzie XMS, natomiast /E wykorzystuje standard EMS 4.0. W drugim przypadku, jeśli tylko tablica symboli przekracza 48 KB należy dokonać redukcji, np. nie specyfikując ścieżki podczas kompilacji lub też używać /Zd zamiast /Zi, gdzie to tylko możliwe. CodeView automatycznie sprawdza kolejno, która z możliwości /X czy /E jest dostępna. Jeśli żadna z nich, to używana jest 64 KB nakładka na dysku (patrz opcja /D). W ten sposób zyskujemy dla programu 64 KB wolnej pamięci.

8. Opcja /2

Opcja ta pozwala na dwumonitorowe śledzenie. Adaptery monitorów muszą być skonfigurowane pod różnymi adresami. Jeden monitor pokazuje wyjście dla aplikacji, natomiast drugi pracę CodeView.

7.6.6.2. Turbo Debugger 2.5

1. Porównanie z BC.EXE

Już na samym początku należy stwierdzić, iż Turbo Debugger jest lepszy od jego odpowiednika w środowisku zintegrowanym, zawiera więcej opcji i umożliwia śledzenie dużych programów. Dodatkowo po konwersji za pomocą TDCONVRT można śledzić programy napisane w Microsoft C 6.0 i skompilowane za pomocą CL /Zi.

2. Funkcje

W oknie podglądania danych można także umieścić nazwę funkcji, wtedy otrzymamy zarazem typy jej parametrów wejściowych, jak i typ danej funkcji. Dzięki ostatniej opcji w menu danych możemy poznać wartość funkcji. Naciskając zaś jednocześnie Alt i klawisz funkcyjny F8 spowodujemy skok do jej końca (instrukcja return).

3. Pliki

Mamy możliwość przeglądania dowolnego zbioru. Oczywiście może to być plik wejściowy, z którego są czytane dane dla programu. W innym oknie może znajdować się plik wyjściowy, obrazujący nowo tworzony zbiór. Mamy wtedy całkowitą kontrolę nad operacjami czytania i zapisywania.

4. Pułapki

Turbo Debugger udostępnia kilka, niedostępnych w środowisku zintegrowanym, rodzajów pułapek, np. przerwanie wykonania po zmianie wartości zmiennej, lub gdy znajdzie się ona w zadanym przedziale. Otrzymamy ten wynik za pomocą opcji Expression true global. Natomiast w opcji Changed memory global specyfikujemy interesujący nas początek obszaru i po przecinku liczbę sprawdzanych jednostek. Podobnie jak w CodeView można ustawić krotność pomijania danej pułapki, program przerywa działanie po kilkakrotnym przejściu przez punkt kontrolny (opcja Pass Count w lokalnym menu pułapek). W komputerach bazujących na procesorze 80386(486), po zainstalowaniu drivera TDH386.SYS i stosowaniu pułapek sprzętowych nie uwidacznia się już opóźnienie powodowane przez pułapki programowe.

5. Zdalne śledzenie

Tak jak w przypadku Turbo Profilera możliwe jest zdalne śledzenie programów. Nie zmieniono też schematu postępowania, ani opcji z linii zlecenia odpowiednika TFREMOTE programu TDREMOTE. Dlatego krótki opis TFREMOTE w p. 7.5.5 jest także wystarczający dla TDREMOTE.

6. Programy rezydentne i obsługi urządzeń zewnętrznych

Wielką zaletą wersji TD 2.0 i późniejszych jest możliwość śledzenia programów rezydentnych (TSR) i programów obsługi urządzeń zewnętrznych, popularnie zwanych device driverami. Dokładny opis przekracza ramy tej publikacji, ale wszelkie potrzebne informacje można znaleźć w pracy [BC5].

7. Praca wirtualna

Podobnie jak TF386 (patrz p. 7.5.5) instaluje się TD386, który pozwala w pełni wykorzystać procesor 32-bitowy i pamięć znajdującą się powyżej 1 MB bez utraty szybkości. Sam proces śledzenia nie różni się niczym od normalnego trybu pracy z TD.

Na koniec wspomnieć należy, że z porównania CodeView z TD wynika, iż drugi program jest łatwiejszy w obsłudze – w prostszy sposób rozwiązano w nim komunikację z użytkownikiem. Do wykonania tych samych operacji wymaga on mniejszej liczby uderzeń w klawisze.

7.6.7. Narzędzia dodatkowe

Debuggery zwykle nie występują jako pojedyncze programy, lecz zazwyczaj w otoczeniu innych programów pomocniczych. W pakiecie Turbo Debuggera są to:

1. Konwerter **TDCONVRT**, który pozwala śledzić programy napisane w języku C i assemblerze, utworzone za pomocą kompilatorów Microsofta.
2. Program przesyłający **TDRF**, pracujący w powiązaniu ze zdalnym śledzeniem i umożliwiający wykonanie podstawowych komend zarządzania w systemie zdalnym.
3. Program **TDSTRIP**, oddzielający tablicę symboli od programu wynikowego bez potrzeby powtórnego łączenia.
4. Program **TDPACK**, pakujący informacje dla debuggera (tablicę symboli).
5. Program **TDMAP**, dołączający do kodu wynikowego .EXE informacje dla debuggera z pliku .MAP, pozwalając w ten sposób śledzić dowolny program.

6. Program **TDDEV**, wyświetlający informacje o wszystkich rezydentnych driverach (programach obsługi dla urządzeń zewnętrznych).
7. Program **TDMEM**, wyświetlający wykorzystanie pamięci w systemie, aktualną tablicę alokacji pamięci, wielkość największego dostępnego bloku łącznie z wykorzystaniem pamięci rozszerzonej typu expanded i extended.
8. Program **TDUMP**, disassemblerujący pliki .EXE, .OBJ i .LIB.
9. Program rezydentny **TDNMI**, umożliwiający śledzenie pracę systemu za pomocą TD przez użytkowników karty Periscope I.

Rozdział 8

ZAŁĄCZNIK

8.1. Ekran

8.1.1. Stałe

Znaki tworzące ramki:

Nazwa znaku	wartość	Nazwa znaku	wartość
C_UL	218	C_UR	191
C_LL	192	C_LR	217
C_XD	194	C_XU	193
C_XR	195	C_XL	180
C_XX	197	C_H	196
C_V	179	C_ULD	201
C_URD	187	C_LLD	200
C_LRD	188	C_XDD	203
C_XUD	202	C_XRD	204
C_XLD	185	C_XXD	206
C_HD	205	C_VD	186
C_HATCH	176	C_SPACE	255

Atrybuty:

A_NORM	0x07
A_INVERSE	0x70
A_INTENSE	0x0F
A_BLINK	0x87
A_UNDER	0x01

Kierunki przesuwania:

DIR_UP	0
DIR_DOWN	1
DIR_LEFT	2
DIR_RIGHT	3

8.1.2. 'Ekran fizyczny

8.1.2.1. Definicje typów i makr

```
#define ROWS 24      /* liczba wierszy ekranu fizycznego */
#define COLS 80      /* liczba kolumn ekranu fizycznego */
#define GIANT 9999

typedef struct s_cell
{
    char chr;
    int att;
} CELL;

typedef struct s_rect
{
    short r1, c1;      /* róg górny lewy */
    short r2, c2;      /* róg dolny prawy */
} RECT;

#define RWIDTH(a) ((a)->c2 - (a)->c1 +1)
#define RHEIGHT(a) ((a)->r2 - (a)->r1 +1)
#define RASG(a,row1,col1,row2,col2) ((a)->r1=row1,(a)->c1=col1,\
    (a)->r2=row2, (a)->c2=col2)
#define RCPY(dst,src) ((dst)->r1=(src)->r1, (dst)->c1=(src)->c1,\
    (dst)->r2=(src)->r2, (dst)->c2=(src)->c2)

#define FALSE 0
#define TRUE 1
#define MIN(x,y) ((x)<(y)? (x):(y))
#define MAX(x,y) ((x)>(y)? (x):(y))
typedef int BOOLEAN;

RECT GIANT_RECT = {0, 0, GIANT, GIANT};

#define MAXROW 24
#define MAXCOL 79
static union
{
    CELL cell;
    short word;
} dspbuf[MAXROW+1][MAXCOL+1];
```

8.1.2.2. Deklaracje funkcji warstwy ekranu fizycznego

```

void PSbegin() /* inicjowanie pracy warstwy ekranu fizycznego */
void PSend() /* zakończenie pracy z warstwą ekranu fizycznego */
void PSsynch() /* synchronizacja zawartości bufora i ekranu */
int PSheight() /* liczba wierszy ekranu fizycznego */
int PSwidth() /* liczba kolumn ekranu fizycznego */
void PSwrite(row, col, ncols, str, att)
    /* wypisywanie ciągu znaków */
int row; /* zmieniany wiersz ekranu */
int col; /* pierwsza zmieniana kolumna */
int ncols; /* maksymalna liczba znaków do wpisania */
char *str; /* zapisywany ciąg */
int att; /* atrybut zapisywanych znaków */
void PSwrtcells(row, col, captr, ncols)
    /* wypisanie ciągu struktur CELL */
int row; /* zmieniany wiersz ekranu */
int col; /* pierwsza zmieniana kolumna */
CELL *captr; /* wskaźnik na ciąg struktur do zapisu na ekran */
int ncols; /* maksymalna liczba kolumn do zapisania */
void PSfill(srect, chr, att)
    /* wypełnianie znakami prostokąta */
RECT *srect; /* wskaźnik na wypełniany prostokąt */
char chr; /* znak */
int att; /* atrybut znaku */
BOOLEAN PSslide(srect, dir, dist)
    /* przemieszczanie prostokąta */
RECT *srect /* przesuwany prostokąt */
int dir /* kierunki: DIR_UP, DIR_DOWN, DIR_LEFT, DIR_RIGHT */
int dist /* odległość, na jaką odbywa się przesunięcie */
void PSsetcur(row, col)
    /* ustawienie kursora */
int row; /* wiersz, w jakim ma być ustawiony kursor */
int col; /* jak wyżej dla kolumny */
void PSshowcur(on)
    /* wyświetlanie/kaśowanie kursora */
BOOLEAN on; /* TRUE lub FALSE odpowiednio dla wyświetlania
             i zgaszenia kursora */

```

```
void PSbeep();  
/* uaktywnienie sygnału dźwiękowego */
```

8.1.3. Okna

8.1.3.1. Definicje typów i makr

```
#define MAXWIND 7  
#define F_UNFRAMED 0  
#define F_AUTO 1  
#define F_SINGLE 2  
#define F_DOUBLE 3  
#define F_BLANK 4  
  
typedef struct  
{  
RECT srect; /* położenie okna na ekranie fizycznym */  
RECT svrect; /* kopia srect (dla powiększania) */  
FRMTYPE frmtype; /* typ ramki */  
short frmatt; /* atrybut ramki */  
char *title; /* tytuł ramki */  
FRMTYPE svfrmtype; /* kopia tytułu (dla powiększania) */  
BOOLEAN zoomed; /* czy okno jest powiększone? */  
short currow; /* położenie kursora: wiersz */  
short curcol; /* jak wyżej: kolumna */  
BOOLEAN showcur; /* czy kursor ma być pokazany? */  
int (*drawfcn)(); /* funkcja odtwarzania zawartości */  
} WINDOW;
```

8.1.3.2. Deklaracje funkcji warstwy okien

```
void Wbegin() /* inicjowanie pracy warstwy okien */  
void Wend() /* zakończenie pracy z warstwą okien */  
int Wnew(redraw) /* utworzenie nowego okna */  
int (*redraw)();  
void redraw(wnum, wrectp)  
/* funkcja odtwarzająca fragment okna */  
int wnum; /* numer okna */  
RECT *wrectp /* odtwarzany fragment okna */
```

```

void Wsetphys(wnum, srectp)
    /* zmiana rozmiarów okna */
int wnum;          /* numer okna */
RECT *srectp;      /* współrzędne okna */

void Wsetfrm(wnum, frmtype, frmatt, title)
    /* określenie typu ramki */
int wnum;          /* numer okna */
FRMTYPE frmtype;   /* rodzaj ramki */
int frmatt;        /* atrybut ramki */
char *naglowek;    /* nagłówek ramki */

void Wdispose(wnum)
    /* kasowanie okna */
int wnum;          /* numer usuwanego okna */

void Wwrite(wnum, wrow, wcol, ncols, str, att)
    /* wypisywanie ciągu znaków */
int wnum;          /* numer okna */
int wrow;          /* zmieniany wiersz okna */
int wcol;          /* pierwsza zmieniana kolumna */
int ncols;         /* maksymalna liczba znaków do wpisania */
char *str;         /* zapisywany ciąg */
int att;           /* atrybut zapisywanych znaków */

void Wwrtcells(wnum, row, col, captr, ncols)
    /* wypisywanie ciągu struktur CELL */
int wnum;          /* numer okna */
int row;           /* zmieniany wiersz okna */
int col;           /* pierwsza zmieniana kolumna */
CELL *captr;       /* wskaźnik na ciąg struktur do zapisu na ekran */
int ncols;         /* maksymalna liczba znaków do wpisania */

void Wfill(wnum, wrectp, chr, att)
    /* wypełnienia prostokąta wewnątrz okna */
int wnum;          /* numer okna */
RECT *wrectp;      /* wskaźnik na wypełniany prostokąt */
char chr;          /* znak */
int att;           /* atrybut znaku */

BOOLEAN Wslide(wnum, wrectp, dir, dist)
    /* przemieszczanie prostokąta wewnątrz okna */
int wnum;          /* numer okna */
RECT *wrectp       /* przesuwany prostokąt */
int dir            /* dozwolone są 4 podstawowe kierunki:

```

```

        DIR_UP, DIR_DOWN, DIR_LEFT, DIR_RIGHT
        zdefiniowane w pliku nagłówkowym display.h */
int dist          /* odległość, na jaką odbywa się przesunięcie */
void Wsetcur(int wnum, row, col)
                /* ustawianie kursora w oknie */
int wnum;        /* numer okna */
int row;         /* wiersz, w którym ma być ustawiony kursor */
int col;         /* jak wyżej dla kolumny */
void Wshowcur(int wnum, on)
                /* kasowanie/pokazywanie kursora */
int wnum;        /* numer okna */
BOOLEAN on;      /* TRUE lub FALSE odpowiednio dla wyświetlenia
                i zgaszenia kursora */
void Wtop(wnum)  /* przesunięcie na szczyt okna */
int wnum;        /* numer okna */
int Wgettop()    /* podaje numer wierzchniego okna */
void Wshuffe()   /* przesunięcie okna ze szczytu na dół stosu */
void Whide(wnum) /* schowanie okna */
int wnum;        /* numer okna */
void Wzoom(int wnum)
                /* powiększenie okna do rozmiarów całego ekranu */
void Wunzoom(int wnum)
                /* przywrócenie "normalnych" rozmiarów okna */
BOOLEAN Wiszoomed(int wnum)
                /* czy okno jest powiększone? */

```

8.1.4. Ekran y wirtualne

8.1.4.1. Definicje typów i makr

```

#define MAXVSCREEN MAXWIND
typedef struct
{
short winnum;    /* numer okna przypisanego do ekranu */
short vwinrow1;  /* położenie okna na ekranie, wiersz */
short vwincol1;  /* jak wyżej dla kolumny */
short vrow2;     /* liczba wierszy ekranu wirtualnego */
short vcol2;     /* liczba kolumn ekranu wirtualnego */
short vcurrow;   /* położenie kursora na ekranie, wiersz */

```

```

short vcurcol;      /* jak wyżej dla kolumny */
CELL **ca;          /* wskaźnik na bufor z zawartością ekranu */
) VSCREEN;

```

8.1.4.2. Deklaracje funkcji warstwy ekranów wirtualnych

```

void VSbegin();      /* początek pracy warstwy ekranów wirtualnych */
void VSend();        /* koniec pracy warstwy ekranów wirtualnych */
int VSnew(nrows, ncols, srectp, title, att)
/* tworzenie nowego ekranu wirtualnego */
int nrows;           /* liczba wierszy */
int ncols;           /* liczba kolumn */
RECT *srectp;        /* położenie i rozmiar okna */
char *title;         /* nagłówek okna */
int att;             /* atrybut wyświetlania nagłówka okna */
void VSdispose(vnum)
/* kasowanie ekranu wirtualnego */
int vnum;            /* numer usuwanego ekranu */
void VSgetsize(vnum, nrowsp, ncolsp)
/* określenie rozmiarów ekranu wirtualnego */
int vnum;            /* numer ekranu wirtualnego */
int *nrowsp;         /* liczba wierszy ekranu */
int *ncolsp;         /* liczba kolumn ekranu */
void VSwrite(vnum, vrow, vcol, ncols, str, att)
/* wypisywanie ciągu znaków */
int vnum;            /* numer ekranu wirtualnego */
int vrow;            /* numer wiersza */
int vcol;            /* numer kolumny */
int ncols;           /* długość ciągu znaków */
char *str;           /* wypisywany ciąg znaków */
int att;             /* atrybut wypisywanego ciągu */
void VSfill(vnum, vrectp, char chr, int att)
/* wypełnianie prostokąta */
int vnum;            /* numer ekranu wirtualnego */
RECT *vrectp;        /* prostokąt do wypełnienia */
char chr;            /* znak wypełniający prostokąt */
int att;             /* atrybut znaku chr */

```

```

void VSslide(vsnun, vrectp, dir, dist)
    /* przemieszczanie prostokąta */
int vsnun;      /* numer ekranu wirtualnego */
RECT *vrectp;   /* prostokąt do przesunięcia */
int dir;        /* kierunek przesunięcia */
int dist;       /* odległość */

void VSsetcur(vsnun, vrow, vcol)
    /* ustawianie położenia kursora */
int vsnun;      /* numer ekranu wirtualnego */
int vrow;       /* wiersz */
int vcol;       /* kolumna */

void VSshowcur(vsnun, on)
    /* wyświetlenie/gaszenie kursora */
int vsnun;      /* numer ekranu wirtualnego */
BOOLEAN on;     /* zapal/zgaś kursor */

void VSgetcur(vsnun, vrowp, vcolp)
    /* ustalenie położenia kursora */
int vsnun;      /* numer ekranu wirtualnego */
int *vrowp;     /* wiersz, w którym znajduje się kursor */
int *vcolp;     /* kolumna, w której znajduje się kursor */

void VSgetwloc(vsnun, vrowlp, vcollp)
    /* ustalenie położenia okna w ramach ekranu
       wirtualnego */
int vsnun;      /* numer ekranu wirtualnego */
int *vrowlp;    /* położenie lewego górnego wiersza */
int *vcollp;    /* położenie lewej górnej kolumny */

void VSpan(vnum, dir, dist)
    /* przesuwanie okna po ekranie wirtualnym */
int vsnun;      /* numer ekranu wirtualnego */
int dir;        /* kierunek przemieszczenia */
int dist;       /* odległość */

int VSgetwnum(vsnun)
    /* ustalenie numeru okna przypisanego ekranowi */
int vsnun;      /* numer ekranu wirtualnego */

int VSgetvsnun(int wnum)
    /* ustalenie numeru ekranu związanego z oknem */
int wnum;       /* numer okna */

```

8.1.5. Deklaracje funkcji warstwy funkcji użytkowych

```

void VSstart()      /* inicjowanie pracy warstwy funkcji użytkowych */
int VSpokaz_linie(nagl, txt, r1)
                    /* edycja pojedynczej linii tekstu */
char *nagl;         /* nagłówek tekstu */
char *txt;          /* przeglądany tekst */
RECT *r;            /* okno, w którym tekst jest przeglądany */
void VSzmien_okno()
                    /* zmiana rozmiarów i położenia okna ekranu
                     wirtualnego */
int vs;             /* numer ekranu wirtualnego */
int VSmenu(txt, nagl, wybor, r) /* wybór jednego z elementów */
char *txt[];        /* tablica wskaźników wybieranych elementów;
                     koniec tablicy sygnalizuje wskaźnik = NULL */
char *nagl;         /* nagłówek okna */
int wybor;          /* numer początkowo wybranego elementu */
RECT *r;            /* rozmiar i położenie okna dla ekranu wirtualnego
                     */

```

8.2. Klawiatura

8.2.1. Deklaracje funkcji

```

void Kbegin(void)   /* początek pracy z klawiaturą */
void Kend(void)     /* koniec pracy z klawiaturą */
int Kget(void)      /* pobranie kodu z klawiatury */
BOOLEAN Kready(void) /* czy jest znak do pobrania */
void Ksynch()       /* warunkowa synchronizacja ekranu */
static BOOLEAN compile()
                    /* wczytywanie definicji kodów wirtualnych
                     klawiatury */

```

8.2.2. Tablica kodów wirtualnych

```

# Kody wirtualne klawiatury
# Sekwencje rozpoczynające się +0 odpowiadają klawiaturze IBM PC
127 +0 83   # Skasuj
301 +0 72   # Kursor w górę

```

```
301 ^e
302 +0 80 # Kursor w dół
302 ^x
303 +0 77 # Kursor w prawo
303 ^d
304 +0 75 # Kursor w lewo
304 ^s
305 +0 73 # Strona w górę
305 ^r
306 +0 81 # Strona w dół
306 ^c
307 +0 119 # Ctrl Home
307 ^t
308 +0 117 # Ctrl End
308 ^y
309 +0 71 # Home
309 ^w
310 +0 79 # End
310 ^z
311 +0 82 # Ins
311 ^v
312 +0 115 # Ctrl-(-
312 ^a
313 +0 116 # Ctrl-->
313 ^f
315 +0 15 # Shift-Tab
315 ^b
321 ^u # Przewiń okno w górę
322 ^j # Przewiń okno w dół
323 ^l # Przewiń okno w prawo
324 ^k # Przewiń okno w lewo
325 +0 74 # Zmniejsz
325 ^o
326 +0 78 # Powiększ
326 ^p
331 +0 59 # F1
331 ^q 1
*****
339 +0 67 # F9
339 ^q 9
340 +0 68 # F0
340 ^q 0
```

```

371 +0 30  # Alt-A
371 ^q ^a
*****
396 +0 44  # Alt-Z
396 ^q ^z
# Tylko dla IBM PC
341 +0 84  # Shift-F1
*****
350 +0 93  # Shift-F10
351 +0 94  # Ctrl-F1
*****
360 +0 103 # Ctrl-F10
361 +0 104 # Alt-F1
*****
370 +0 113 # Alt-F10
# Znaki specjalne przetwarzane literalnie
^@ ^^ ^@  # Ctrl-@
^a ^^ ^a  # Ctrl-a
*****
^z ^^ ^z  # Ctrl-z
^[ ^^ ^[  # Ctrl-[
^^ ^^ ^^  # Ctrl-^
^] ^^ ^]  # Ctrl-]
^^ ^^ ^^  # Ctrl-^
^_ ^^ ^_  # Ctrl-_
# Koniec tablicy definicji klawiatury

```

8.3. Przetwarzanie tekstu

8.3.1. Baza rekordów zmiennej długości

```

int dzd_ini(w, dl_rek, nazwa)
/* zakładanie bazy dzd */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */
unsigned dl_rek; /* długość "fizycznego" rekordu w bazie */
char *nazwa /* nazwa bazy */

int dzd_open(w, nazwa)
/* otwieranie istniejącej bazy dzd */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */
char *nazwa /* nazwa otwieranej bazy */

```

```
int dzd_close(w) /* zamykanie bazy dzd */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */

long dzd_ins(w, dl, tresc)
/* dopisanie nowego rekordu, wartością
funkcji jest identyfikator nadany nowo
wprowadzanemu rekordowi */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */
unsigned dl; /* długość w bajtach zapisywanych danych */
char *tresc; /* wskaźnik na początek rekordu */

int dzd_get(w, gdzie)
/* wczytanie rekordu z bazy */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */
dzd *gdzie; /* wskaźnik na strukturę opisującą rekord */

int dzd_rep(w, dl, tresc)
/* zmiana treści bieżącego rekordu */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */
unsigned dl; /* długość zmienianych danych */
char *tresc; /* wskaźnik na nową zawartość rekordu */

int dzd_set(w, ident)
/* ustawienie wskaźnika bieżącego rekordu */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */
long ident; /* identyfikator rekordu */

int dzd_fst(ops_dzd *w)
/* ustawienie wskaźnika na pierwszy rekord */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */

int dzd_nxt(w) /* przesunięcie wskaźnika na następny rekord */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */

int dzd_del(w) /* kasowanie bieżącego rekordu */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */

int dzd_err() /* podaj kod ostatniego błędu */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */

char *dzd_err_txt(ops_dzd *w)
/* podaj opis błędu */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */

int dzd_err_clr(ops_dzd *w)
/* kasuj błąd */
ops_dzd *w; /* wskaźnik na strukturę opisującą bazę */
```

8.3.2. Translacja wyrażeń na Odwrotną Notację Polska

8.3.2.1. Definicje typów i makr

```
typedef struct
```

```
{
```

```
char nazwa[MAX_TOKEN];
```

```
char typ;
```

```
} token;
```

Typy tokenów:

```
#define PRAWY_NAWIAS 1
```

```
#define LEWY_NAWIAS 2
```

```
#define OPERATOR 3
```

```
#define TEKST 5
```

```
#define L_CALK 6
```

```
#define L_ZMP 7
```

```
#define KONIEC 10
```

```
typedef (*f_int)();
```

```
typedef struct
```

```
{
```

```
char *nazwa;
```

```
char typ;
```

```
char priorytet;
```

```
f_int f;
```

```
} operator;
```

Typy operatorów

```
#define UNARNY 8
```

```
#define BINARNY 9
```

```
operator teksty[] =
```

```
{
```

```
"*", BINARNY, 10, txt_ilocz,
```

```
"+", BINARNY, 9, txt_suma,
```

```
"dokl", UNARNY, 12, txt_dokl,
```

```
"^", UNARNY, 11, txt_neg,
```

```
NULL,
```

```
};
```

8.3.2.2. Deklaracje funkcji

```
int zamien_na_onp(txt_inf, op, tab, dl_tab)
char *txt_inf;      /* wskaźnik na wyrażenie w zapisie infiksowym */
operator *op;       /* tablica z opisem stosowanych operatorów */
token *tab;         /* tablica na zapisanie wyrażenia w ONP */
int dl_tab;         /* liczba elementów w tablicy tab /

int na_stos(stos, max, typ, token, wolny)
token *stos;        /* tablica z zapisem w ONP */
int max;            /* maksymalna liczba tokenów w tablicy stos */
int typ;            /* typ zapisywanego tokena */
char *token;        /* ciąg znaków reprezentujący token */
int *wolny;         /* wskaźnik na pierwszy wolny element
                    tablicy stos */

int ze_stosu(token *stos, int *typ, char *tok, int *wolny)
token *stos;        /* tablica z zapisem w ONP */
int *typ;           /* typ pobieranego tokena */
char *token;        /* ciąg znaków reprezentujący pobierany token */
int *wolny;         /* wskaźnik na pierwszy wolny element
                    tablicy stos */
```

8.3.3. Lista inwersyjna

8.3.3.1. Definicje typów

```
typedef struct
{
char *termin;       /* wskaźnik na termin indeksowy */
longident;          /* identyfikator z terminu w bazie dzd */
unsigned *dane;      /* tablica identyfikatorów tekstów */
int dl_danych;      /* długość tablicy identyfikatorów */
} ops_ter;

typedef struct
{
char *termin;
int ident;
} typ_ter;
```

```
typedef struct
{
ops_dzd baza;
typ_ter *dane;
unsigned liczba_term;
} ops_inv;
```

8.3.3.2. Deklaracje funkcji

```
int inv_ini(nazwa_pl, terminy, inv)
/* zakładanie nowej tablicy inwersyjnej */
char *nawa_pl; /* nazwa tworzonej tablicy inwersyjnej */
char *terminy[]; /* terminy do umieszczenia w tablicy */
ops_inv *inv; /* wskaźnik na strukturę opisującą tablicę */

int inv_open(nazwa_pl, inv)
/* otwieranie istniejącej tablicy inwersyjnej */
char *nawa_pl; /* nazwa tworzonej tablicy inwersyjnej */
ops_inv *inv; /* wskaźnik na strukturę opisującą tablicę */

int inv_close(inv)
/* zamykanie używanej tablicy inwersyjnej */
ops_inv *inv; /* wskaźnik na strukturę opisującą tablicę */

int inv_ins(inv, dane)
/* dopisanie nowego terminu do listy inwersyjnej */
ops_inv *inv; /* wskaźnik na strukturę opisującą tablicę */
ops_ter *dane; /* wskaźnik na strukturę opisującą element */

int inv_get(inv, dane)
/* pobranie elementu tablicy inwersyjnej */
ops_inv *inv; /* wskaźnik na strukturę opisującą tablicę */
ops_ter *dane; /* wskaźnik na strukturę opisującą element */

int inv_del(inv, dane)
/* kasowanie terminu umieszczonego w tablicy
inwersyjnej */
ops_inv *inv; /* wskaźnik na strukturę opisującą tablicę */
char *termin; /* nazwa dopisywanego elementu */

int inv_rep(inv, dane)
/* zmiana treści elementu tablicy inwersyjnej */
ops_inv *inv; /* wskaźnik na strukturę opisującą tablicę */
ops_ter *dane; /* wskaźnik na strukturę opisującą element */
```

```

int inv_add(inv, termin, dopisz);
                /* dopisanie nowego identyfikatora do elementu
                tablicy inwersyjnej */
ops_inv *inv;    /* wskaźnik na strukturę opisującą tablicę */
char *termin;    /* nazwa dopisywanego elementu */
unsigned dopisz;  /* dodawany identyfikator tekstu */

```

8.4. Komendy edytorów programowych

Opis komendy	Brief	Borland
Podstawowe:		
Początek wiersza	Home	Home
Koniec wiersza	End	End
Początek okna	Home Home	Ctrl-Home
Koniec okna	End End	Ctrl-End
Początek pliku	Home Home Home	Ctrl-PgUp
Koniec pliku	End End End	Ctrl-PgDn
Słowo w prawo	Ctrl-Prawo	Ctrl-Prawo, Ctrl-F
Słowo w lewo	Ctrl-Lewo	Ctrl-Lewo, Ctrl-A
Kasowanie:		
Kasowanie znaku przed kursorem	Bs	Bs
Kasowanie znaku pod kursorem	Del	Del
Kasowanie do końca wiersza	Alt-K	Ctrl-Q Y
Kasowanie wiersza	Alt-D	Ctrl-Y
Bloki:		
Zaznacz blok	Alt-C, Alt-L, Alt-M	Shift-kursor, Ctrl-K B(K)
Skopiuj blok	Num +, Ins	Ctrl-K C, Ctrl-Ins
Przesuń blok	Num -, Ins	Ctrl-K V, Shift-Del
Skasuj blok	Num -	Ctrl-K Y, Ctrl-Del
Drukuj blok	Alt-P	Ctrl-K P
Skopiuj blok pod kursor	Ins	Shift-Ins
Okna:		

Utworzenie okna	F3 Strzałka	Alt-F N
Kasowanie okna	F4	Alt-F3
Następny(e) plik(okno)	Alt-N	F6
Powiększenie (ZOOM)	Alt-F2	F5
Zmiana okna	F1 Strzałka	Alt-(1-9)
Zmiana rozmiarów	F2 Strzałka	Ctrl-F5
Zamknięcie aktywnego okna	F4	Alt-F3

Pliki:

Edycja pliku	Alt-E	F3, Alt-F O
Zapamiętanie pliku	Alt-W	F2, Alt-F S
Lista plików	Alt-B	Alt-O
Zmiana nazwy pliku	Alt-O	Alt-F A

Wzorzec:

Szukanie wzorca	F5	Ctrl-Q F, Alt-S F
Szukanie wzorca w przód	Alt-S	
Szukanie wzorca w tył	Alt-F5	
Zamiana wzorca	F6	Ctrl-Q A, Alt-S R
Zamiana wzorca w przód	Alt-T	
Zamiana wzorca w tył	Alt-F6	
Powtórzenie wyszukiwania	Shift-F5	Ctrl-L, Alt-S S
Powtórzenie podmieniania	Shift-F6	

Pozostałe komendy:

Tryb Insert	Alt-I	Ctrl-V, Ins
Centruj wiersz	Ctrl-C	
System Dos	Alt-Z	Alt-F D
Koniec	Alt-X	Alt-X, Alt-F Q
Znacznik miejsca	Alt-(0-9)	Ctrl-K (0-9)
Skok do znacznika	Alt J	Ctrl-Q (0-9)
Skok do wiersza	Alt-G	Alt-S G
Anulowanie	Num. *, Alt-U	Alt-E U, Alt-Bs

Pomocnik	Alt-H	F1
Zapamiętaj sekwencję	F7	
Odtwórz sekwencję	F8	
Wczytaj makro klawiatury	Alt-F7	
Zapamiętaj makro	Alt-F8	
Wykonaj komendę	F10	
Powtórz operację	Ctrl-R	
Sprawdzenie nawiasów	F10 brace	
Kompilacja wewnętrzna	Alt-F10	Alt-C
Następny błąd	Ctrl-N	Alt-F8
Poprzedni błąd		Alt-F7
Lista błędów	Ctrl-P	

8.5. Priorytety i sposób łączenia operatorów

Priorytet	Łączenie	Operator	Nazwa operatora
1.	Lewe	()	Nawiasy okrągłe
		[]	Nawiasy kwadratowe
		->	Operator strzałkowy
		.	Operator kropkowy
2.	Prawe	!	Logiczna negacja
		~	Bitowe dopełnienie
		+	Unarny plus
		-	Unarny minus
		++	(Pre) postinkrementacja
		--	(Pre) postdekrementacja
		&	Adres
		*	Wskazanie
3.	Lewe	sizeof	Rozmiar
		*	Mnożenie
		/	Dzielenie
		%	Reszta z dzielenia
4.	Lewe	+	Dodawanie
		-	Odejmowanie
5.	Lewe	<<	Przesunięcie w lewo
		>>	Przesunięcie w prawo
6.	Lewe	<	Mniejszy niż
		<=	Mniejszy niż lub równy

		>	Większy niż
		>=	Większy niż lub równy
7.	Lewe	==	Równy
		!=	Nierówny
8.	Lewe	&	Bitowy AND
9.	Lewe	^	Bitowy XOR
10.	Lewe		Bitowy OR
11.	Lewe	&&	Logiczny AND
12.	Lewe		Logiczny OR
13.	Prawe	?:	Wybór
14.	Prawe	=	Przypisanie proste
		*=	Przypisz iloczyn
		/=	Przypisz iloraz
		%=	Przypisz resztę z dzielenia
		+=	Przypisz sumę
		-=	Przypisz różnicę
		&=	Przypisz bitowy AND
		^=	Przypisz bitowy XOR
		=	Przypisz bitowy OR
		<<=	Przypisz przesunięcie w lewo
		>>=	Przypisz przesunięcie w prawo
15.	Lewe	,	Połącz

BIBLIOGRAFIA

- [ADV] ROCHKIND M.J., *Advanced C programming for displays*, NJ, Prentice Hall Software Series, Englewood Cliffs, 1988.
- [ALGinC] SEDGEWICK R., *Algorithms in C*, Addison-Wesley Publishing Company, Reading Ma, 1990.
- [ANSI] *Draft programming language C*, ANSI, 1986.
- [BC1] Borland C++ version 2.0 - user's guide, Borland International, Green Hills, California, 1991.
- [BC2] Borland C++ version 2.0 - programmer's guide, Borland International, Green Hills, California, 1991.
- [BC3] Turbo Profiler version 1.1 - user's guide, Borland International, Green Hills, California, 1991.
- [BC4] Borland C++ version 2.0 - library reference, Borland International, Green Hills, California, 1991.
- [BC5] Turbo Debugger version - user's guide, Borland International, Green Hills, California, 1991.
- [Biel] BIELECKI J., *Turbo C dla programistów*, Warszawa, WKŁ, 1989.
- [ISIS] *Mini-mikro CDS/ISIS reference manual (version 2.3)*, UNESCO, Paryż, 1989.
- [KR] KERNIGHAM B.W., RITCHIE D.M., *Język C*, Warszawa, WNT, 1988.
- [MC] *Microsoft C 6.0 advanced programming techniques*, Microsoft, 1990.
- [Niedz] NIEDZIELSKA E. (red.), *Informatyka. Technologia przetwarzania danych*, Warszawa, PWE, 1983.
- [Schil] SCHILDT H., *Turbo C the complete reference*, Berkeley, Borland Osborne/McGraw Hill, 1988.
- [Wirth] WIRTH N., *The programming language Pascal*, Acta Informatica, Vol. 1, No. 1, 1971, s. 35-63.

SKOROWIDZ

A

Alokacja pamięci 23,113

B

Błędy 170

"Null pointer assignment"

172

argument funkcji 173

brak dealokacji zmiennych

177

brak deklaracji 173

brak kodu 180

brak instrukcji return 180

break 180

efekt uboczny 176,179

kolejność wykonywania

operacji 171

niezainicjowane zmienne 177

ostatnie przejście w pętli

177

podobieństwo operatorów 178

propagacja bitu znaku 178

przepelnienie stosu 174

przetawienie średników 179

przesłanianie zmiennych 179

redukcja bitu znaku 178

usuwanie 170

wskaźnik 172,178

zmienna lokalna wynikiem

funkcji 179

D

Debugger 180

CodeView 3.0 181

dwumonitorowe śledzenie 183

obsługa urządzeń zewnętrznych
184

pamięć 182

praca wirtualna 184

programy rezydentne 184

pułapka 182

TDCONVRT 184

TDDEV 185

TDMAP 184

TDMEM 185

TDPACK 184

TDRF 184

TDSTRIP 184

TDUMP 185

Turbo Debugger 2.5 183

śledzenie zdalne 183

wielookiennosc 181

wskaźnik 181

wydruck kontrolny 182

wywoływanie funkcji 181

zmienna rejestrowa 182

Definicja

funkcji 12,34

Deklaracja

funkcji 12,36

E

Edytor

- ogólnego przeznaczenia 155
- programowy 155,201

Ekran 52

- atrybut 186
- CELL 55,187
- definicja stałych 186
- dspbuf 57,187
- fizyczny 33,187,188
- funkcja użytkowa 194
- intersect 56
- kursor 186
- pokaz_norma 91
- PSbeep 62,189
- PSbegin 59,188
- PSend 59,188
- PSfill 61,64,188
- PSheight 59,188
- PSsetcur 62,188
- PSshowcur 62,188
- PSslide 61,188
- PSsynch 59,188
- PSwidth 59,188
- PSwrite 60,63,188
- PSwrtcells 60,188
- RECT 55,187
- redraw 86
- szukaj_litery 94
- VSbegin 80,192
- VSCREEN 84,192
- VSdispose 81,192
- VSend 80,192
- VSfill 81,192
- VSgetcur 193
- VSgetsize 81,192
- VSgetvsnun 83,193
- VSgetwloc 83,193
- VSgetwnun 83,193
- VSmenu 93,194
- VSnew 80,192
- VSpan 83,193

- VSpokaz_linie 89,194

- VSsetcur 193

- VSshow_cur 193

- VSSlide 82,193

- VStart 88,194

- VSwrite 81,85,192

- Vszmien_okno 92,194

- wirtualny 33,79,191,192

- Wrtscrn 57

G

- Grep 161

- wyrażenie regularne 162

I

- Inkrementalna

- kompilacja 48

- łączenie 48

- Iteracja 175

K

- Klawiatura 96,194

- automat skończony 109

- BKavail 97

- BKread 97

- BKstatus 97

- cget 97

- compare 115

- compile 114,194

- cready 98

- doline 116

- Kbegin 98,194

- kc_table.c 112

- Kend 98,194

- keycode 99,103,107,114

- keycode.h 100

- KEYSEQ 113

- Kget 99,194

- Kready 99

- Keynch 99,194

- lookup 118

- setraw 98

stała mapa wirtualna 102
 tablica definicyjna 108
 tablica kodów wirtualnych
 194
 unsetraw 98
 wirtualna 99
 zewnętrzna mapa wirtualna
 107
 Kod wirtualny 108
 Kursor
 BVsetcpos 58
 BVsetctype 58
 VGetcur 82
 VSsetcur 82
 VShowcur 82
 Wsetcur 71
 Wshowcur 71
 L
 Lib 160
 M
 Make 156
 Model pamięci
 Compact 24
 Huge 24
 Large 24
 Medium 24
 Small 24
 Tiny 24
 Modyfikator
 far 25
 huge 25
 near 25
 Monitorowanie programu
 patrz Profilowanie programu
 N
 Nagłówek pliku programowego 37

O
 Okno 33,67,189
 allocmask 75
 freemask 75
 redraw 69,189
 reset 76
 setmask 75
 Wbegin 68,189
 Wdispose 70,190
 Wend 68,189
 Wfill 71,190
 Wgettop 72,191
 Whide 72,191
 WINDOW 73,189
 Wiszoomed 72,191
 Wnew 68,189
 Wsetcur 191
 Wsetfrm 69,190
 Wsetphys 69,190
 Wshowcur 191
 Wshuffle 72,191
 Wslide 71,190
 Wtop 71,191
 Wunzoom 72,191
 Wwrite 70,190
 Wwrtcells 70,190
 Wzoom 72,191
 Oprogramowanie wspomagające 155
 Optymalizacja 41
 inkrementacja i dekrementacja 42
 prekompilacja 47
 tablica 44
 wskaźnik 44
 wywołanie funkcji 47
 zrzuć rejestrów 43
 P
 Plik 25
 nagłówkowy 34
 programowy 36

- Predefiniowane makra 22
- Prekompilacja 20
 - optymalizacja 47
- Preprocesor 13
 - #define 13
 - #error 16
 - #if, #else, #elif, #endif 16
 - #ifdef, #ifndef, #undef 17
 - #include 16,35
 - #line 19
 - #pragma 19
- Priorytety operatorów 203
- Profiler
 - patrz Profilowanie programu
- Profilowanie programu 163
 - analiza pasywna 166
 - definicja 163
 - interpretacja wyników 168
 - obszar profilowany 165
 - poprawianie programu 168
 - wirtualne 170
 - zator 164
- Przenośność 48
 - system operacyjny 49
 - wielkość standardowych danych 50
- Przetwarzanie tekstu 122,196
- R**
 - Rekord zmiennej długości 123
 - Rekurencja 175
- S**
 - Struktura
 - fizyczna systemu 34
 - logiczna oprogramowania 32
- T**
 - Tekst 132,196
 - algorytm Boyer-Moore 142
 - daj_token 139
 - ini_pomin 143
 - inv_add 147,201
 - inv_close 145,200
 - inv_del 146,200
 - inv_get 146,150,200
 - inv_ini 145,148,200
 - inv_ins 146,200
 - inv_open 145,149,200
 - inv_rep 146,200
 - liczba_wystapien 152
 - lista inwersyjna 199
 - na_stos 135,199
 - Odwrotna Notacja Polska 134
 - operator 136,198
 - ops_inv 147,200
 - ops_ter 144,199
 - podob 152
 - podobienstwo ciagów 151
 - przeszukiwanie 141
 - szukanie wzorca 141,161
 - szybkie_szukanie 143
 - tablica inwersyjna 144
 - tekst_nazwa 140
 - ter_dzd 149
 - token 136,198
 - typ_ter 147,199
 - wolne_szukanie 141
 - wyrażenie
 - elementarne 133
 - nawiasowe 134
 - zamien_na_onp 135,137,199
 - ze_stosu 135,199
 - Tf386 169
 - Tfinst 169
 - Tfremote 169
 - Tworzenie bibliotek 160
 - W**
 - Warstwa DZD 123,196
 - dzd 126
 - dzd_close 124,197
 - dzd_del 125,197
 - dzd_err 125,197

dzd_err_clr 126,197
dzd_err_txt 125,197
dzd_fst 125,197
dzd_get 124,129,197
dzd_ini 123,128,196
dzd_ins 124,197
dzd_nxt 125,197
dzd_open 123,196
dzd_rep 124,197
dzd_set 124,197
Odwrotna Notacja Polska 198
ops_dzd 126

wolny_blok 129
Wielowarstwowe programowanie 33

Z

Zarządzanie

pamięcią 23,113

programem 156

Zmienna

wewnętrzna 10

zewnętrzna 10

Zmienna liczba argumentów funkcji

28

SPIS TRESCI

Rozdział 1. WSTĘP	3
1.1. Dlaczego wybrano język C?	4
1.2. Materiały źródłowe	6
1.3. Przegląd treści	6
1.4. Sposób prezentowania oprogramowania	8

Rozdział 2. WYBRANE ELEMENTY JEZYKA C	9
2.1. Własności zmiennych i funkcji	9
2.1.1. Zmienne	10
2.1.2. Funkcje	12
2.2. Preprocesor	13
2.2.1. #define	13
2.2.2. #error	16
2.2.3. #include	16
2.2.4. #if, #else, #elif i #endif	16
2.2.5. #ifdef, #ifndef i #undef	17
2.2.6. #line	19
2.2.7. #pragma	19
2.2.8. Predefiniowane makra kompilatora	22
2.3. Zarządzanie pamięcią	23
2.4. Obsługa plików	25
2.5. Zmienna liczba argumentów funkcji	28

Rozdział 3. ZASADY OPROGRAMOWYWANIA SYSTEMÓW INFORMACYJNYCH	31
---	----

3.1. Struktura logiczna oprogramowania	32
3.2. Struktura fizyczna systemu	34
3.2.1. Pliki nagłówkowe	34
3.2.2. Pliki programowe	36
3.2.2.1. Nagłówek pliku programowego	37
3.2.2.2. Typy funkcji	37
3.2.2.3. Testowanie pliku programowego	38
3.2.2.4. Przykład	39
3.3. Optymalizacja	41
3.3.1. Inkrementacja i dekrementacja	42
3.3.2. Zmienne rejestrowe	43
3.3.3. Wskaźniki i tablice	44
3.3.4. Wywoływanie funkcji	47
3.3.5. Prekompilowane pliki nagłówkowe	47
3.3.6. Inkrementalna kompilacja i łączenie	48
3.4. Przenośność oprogramowania	48
3.4.1. Uzależnienia od systemu operacyjnego	49
3.4.2. Różnice w wielkości standardowych danych	50
Rozdział 4. EKRAN	52
4.1. Wprowadzenie	52
4.2. Implementacja funkcji obsługi ekranu	53
4.2.1. Wymagania ogólne	53
4.2.2. Podstawowe makra i funkcje dla operacji na ekranie ...	54
4.2.3. Implementacja IBM_mem	56
4.3. Ekran fizyczny	59
4.3.1. Omówienie funkcji	59
4.3.2. Analiza kodu funkcji	62
4.3.3. Przykład	65
4.4. Okna	67
4.4.1. Omówienie funkcji	68
4.4.2. Analiza kodu funkcji	73
4.4.3. Przykład	77

4.5. Ekrany wirtualne	79
4.5.1. Omówienie funkcji	80
4.5.2. Analiza kodu funkcji	84
4.6. Funkcje użytkowe	87
Rozdział 5. Klawiatura	96
5.1. Podstawowe pojęcia	96
5.2. Interfejs dla klawiatury	98
5.3. Klawiatura wirtualna	99
5.3.1. Stała mapa wirtualna	102
5.3.2. Zewnętrzna mapa wirtualna	107
5.3.2.1. Tablica definicyjna dla klawiatury	108
5.3.2.2. Konstrukcja automatu skończonego	109
5.4. Podsumowanie	120
Rozdział 6. PRZETWARZANIE DANYCH TEKSTOWYCH	122
6.1. Przechowywanie danych tekstowych	122
6.1.1. Omówienie funkcji	123
6.1.2. Prezentacja kodu funkcji	126
6.1.3. Przykład	131
6.2. Opis tekstu	132
6.2.1. Syntaktyka i semantyka języka opisu	133
6.2.2. Obliczanie wartości wyrażeń języka opisu tekstów	134
6.2.2.1. Zamiana wyrażenia nawiasowego na Odwrotną Notację Polską	134
6.2.2.2. Analiza kodu funkcji	136
6.3. Przeszukiwanie tekstu	141
6.3.1. Szybkie szukanie wzorca	141
6.3.2. Tablica inwersyjna	144
6.3.2.1. Omówienie funkcji	144
6.3.2.2. Analiza kodu funkcji	147
6.3.3. Określanie podobieństwa ciągów znaków	151
6.3.4. Podsumowanie	153

Rozdział 7. OPROGRAMOWANIE WSPOMAGAJĄCE	155
7.1. Porównanie edytora programowego z edytorem ogólnego przeznaczenia	155
7.2. Zarządzanie programem	156
7.3. Tworzenie bibliotek	160
7.4. Wyszukiwanie wzorców	161
7.5. Profilowanie programu	163
7.5.1. Definicja profilowania	163
7.5.2. Obszary profilowane	165
7.5.3. Monitorowanie programu - analiza aktywna i pasywna ...	166
7.5.4. Interpretacja wyników i poprawianie programu	168
7.5.5. TFREMOTE, TF386 i TFINST	169
7.6. Usuwanie błędów w programach	170
7.6.1. Kolejność wykonywania operacji	171
7.6.2. Problemy ze wskaźnikami	172
7.6.3. Brak deklaracji funkcji i błędne podanie argumentów ..	173
7.6.4. Przpełnienie stosu	174
7.6.5. Błędy spotykane podczas programowania w języku C	176
7.6.6. Stosowanie debuggera	180
7.6.6.1. CodeView 3.0	181
7.6.6.2. Turbo Debugger 2.5	183
7.6.7. Narzędzia dodatkowe	184
Rozdział 8. ZAŁĄCZNIK	186
8.1. Ekran	186
8.1.1. Stałe	186
8.1.2. Ekran fizyczny	187
8.1.2.1. Definicje typów i makr	187
8.1.2.2. Deklaracje funkcji warstwy ekranu fizycznego	188
8.1.3. Okna	189
8.1.3.1. Definicje typów i makr	189
8.1.3.2. Deklaracje funkcji warstwy okien	189

8.1.4. Ekrany wirtualne	191
8.1.4.1. Definicje typów i makr	191
8.1.4.2. Deklaracje funkcji warstwy ekranów wirtualnych	192
8.1.5. Deklaracje funkcji warstwy funkcji użytkowych.....	194
8.2. Klawiatura	194
8.2.1. Deklaracje funkcji	194
8.2.2. Tablica kodów wirtualnych	194
8.3. Przetwarzanie tekstu	196
8.3.1. Baza rekordów zmiennej długości	196
8.3.2. Translacja wyrażeń na Odwrotną Notację Polską	198
8.3.2.1. Definicje typów i makr	198
8.3.2.2. Deklaracje funkcji	199
8.3.3. Lista inwersyjna	199
8.3.3.1. Definicje typów	199
8.3.3.2. Deklaracje funkcji	200
8.4. Komendy edytorów programowych	201
8.5. Priorytety i sposób łączenia operatorów	203
BIBLIOGRAFIA	205
SKOROWIDZ	206



272533 L/

Publikacje w sprzedaży

- Architektura i technologia systemów informatycznych ISAT 91. Information Systems Architecture and technologies ISAT 91* (praca zbior. pod red. M. Bazewicza), Wrocław 1991
- Bilski E., Dubielewicz I., *Model Odniesienia dla Współdziałania Systemów Otwartych*. Tom I, Wrocław 1991
- Huzar A., Mazur U., *Programowanie w języku Pascal* (wyd. II zmienione), Wrocław 1989
- Jacukowicz M., Żak D., *Podstawy użytkowania komputera VAX. Instrukcja użytkowania*, Wrocław 1990
- Kaliś A., *Podstawy teorii układów logicznych*, Wrocław 1989
- Klonowski Z., Paluszkiewicz A., *Programowanie w języku systemu dBASE*, Wrocław 1990
- Koleśnik K., *Ćwiczenia laboratoryjne z języka Pascal*, Wrocław 1991
- Komputerowe systemy sterujące. Ćwiczenia laboratoryjne* (pr. zbior. pod red. Z. Bubnickiego), Wrocław 1989
- Koszalka L., Kurzyński M., *Zbiór zadań i problemów z teorii identyfikacji, eksperymentu i rozpoznawania*, Wrocław 1991
- Mazur H., *Typy danych w programowaniu strukturalnym. Zbiór zadań z programowania w języku Pascal*, Wrocław 1990
- Mikrokomputerowe systemy biblioteczno-informacyjne* (praca zbior. pod red. K. Chorosia), Wrocław 1992
- Mroziński Z., *Elementy systemu mikrokomputerowego serii 8086*, Wrocław 1990
- Mroziński Z., *Programowane układy wspomagające w systemie mikrokomputerowym serii 8080*, Wrocław 1988
- Mroziński Z., *Elementy systemu mikrokomputerowego serii 8080* (wyd. II), Wrocław 1988
- Problemy projektowania, wdrażania i eksploatacji baz danych. Cz. II* (pr. zbior. pod red. J. Dębowskiego), Wrocław 1987
- Protokoły w międzyuczelnianej sieci komputerowej (MSK)* (pr. zbior. pod red. E. Bilskiego), Wrocław 1987
- Slifirska E., Wanik A., *Wybrane zagadnienia z informatyki i metod numerycznych*, Wrocław 1988
- Techniki i metody rozproszonego przetwarzania danych. Cz. II* (pr. zbior. pod red. M. Bazewicza), Wrocław 1988
- Techniki i metody rozproszonego przetwarzania danych. Cz. IV* (pr. zbior. pod red. M. Bazewicza), Wrocław 1989
- Własności i funkcje sieci komputerowych. Cz. III* (pr. zbior. pod red. M. Bazewicza), Wrocław 1983

**Wydawnictwa Politechniki Wrocławskiej
są do nabycia w następujących księgarniach:
„Politechnika”**

**Wybrzeże Wyspiańskiego 27, 50-370 Wrocław
(budynek A-1 PWr.)
oraz Wydawnictwa Politechniki Wrocławskiej
plac Grunwaldzki 13, 50-377 Wrocław
(budynek D-1 PWr.)**