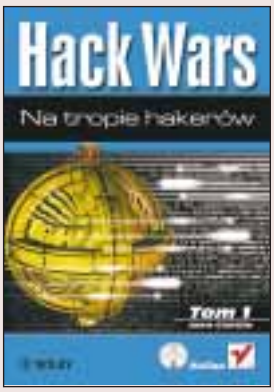
Hack Wars. Tom 1. Na tropie hakerów



Autor: John Chirillo
Tłumaczenie: Paweł Koronkiewicz, Leonard Milcin
ISBN: 83-7197-599-6
Tytuł oryginału: Hack Attacks Revealed: A Complete Reference with Custom Security Hacking Toolkit
Format: B5, stron: 736
Zawiera CD-ROM

Ekspert w dziedzinie zabezpieczeń, John Chirillo, zachęca Czytelnika do poznania mrocznego i tajemniczego świata hakerów. Czerpiąc z bogatego doświadczenia we współpracy z firmami Fortune 1000, Chirillo przedstawia różne sposoby wykorzystania przez hakerów luk w zabezpieczeniach sieci oraz metody rozpoznawania tego rodzaju zagrożeń. Uzupełnieniem jest szczegółowy opis pakietu TigerBox, umożliwiającego hakerom przeprowadzanie skutecznych włamań, a administratorowi sieci – zyskanie pewności, że jest właściwie chroniona.

W tej prowokacyjnej książce znajdziemy:

- Opis protokołów sieciowych i technologii komunikacyjnych z punktu widzenia hakera
- Pełny opis stosowanych metod włamań, wyjaśniający, jak działają hakerzy, crackerzy, "phreaks" i cyberpunki
- Narzędzia do gromadzenia informacji i skanowania sieci, umożliwiające wykrycie i przeanalizowanie przypadków naruszenia bezpieczeństwa systemu
- Dokładne instrukcje, jak posługiwać się pakietem typu TigerBox i wykorzystywać go do wykrywania ataków.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 6.

Podstawy programowania dla hakerów

Język C

Dla każdego hakera, młodego czy starego, mniej lub bardziej doświadczonego, znajomość języka C jest jednym z fundamentów wiedzy. Niemal wszystkie narzędzia i programy, stosowane w trakcie analiz sieci i włamań, powstają właśnie w tym języku. Również w niniejszej książce większość przedstawianego kodu to właśnie kod źródłowy w języku C. Programy te można modyfikować, dostosowywać do własnych potrzeb i odpowiednio kompilować.

W pracy nad niniejszym rozdziałem wykorzystano obszerne fragmenty pracy guru programowania Matthew Proberta. Mają one pełnić funkcję wprowadzenia do programowania w języku C i umożliwić stosowanie przedstawianych w książce (i załączonych na CD-ROM-ie) listingów programów. Pełny kurs języka znajdziesz w niejednej książce wydawnictwa Helion.

Język C wyróżniają następujące cechy, które omawiamy niżej.

- Blokowe konstrukcje sterowania wykonywaniem programu (typowe dla większości języków wysokiego poziomu).
- Swobodne operowanie podstawowymi obiektami „maszynowymi" (takimi jak bajty) i możliwość odwoływania się do nich przy użyciu dowolnej, wymaganej w danej sytuacji, perspektywy obiektowej (typowe dla języków asemblerowych).
- Możliwość wykonywania operacji zarówno wysokiego poziomu (na przykład arytmetyka zmiennoprzecinkowa), jak i niskiego poziomu (zbliżonych do instrukcji języka maszynowego), co umożliwia tworzenie kodu wysoce zoptymalizowanego bez utraty jego przenośności.

Przedstawiony w niniejszym rozdziale opis języka C bazować będzie na funkcjach oferowanych przez większość kompilatorów dla komputerów PC. Powinien dzięki temu umożliwić rozpoczęcie tworzenia prostych programów osobom nieposiadającym szerokiej wiedzy o języku (uwzględnimy między innymi funkcje zapisane w pamięci ROM i funkcje DOS-u).

Przyjmujemy założenie, że masz, drogi Czytelniku, dostęp do kompilatora C i odpowiedniej dokumentacji funkcji bibliotecznych. Programy przykładowe powstały w Turbo C firmy Borland; większość elementów niestandardowych tego narzędzia uwzględniono również w późniejszych edycjach Microsoft C.

Wersje języka C

W pierwotnej edycji języka C (jeszcze przed publikacją Kernighana i Ritchie'ego, *The C Programming Language*, Prentice-Hall 1988 (polskie wydanie: *Język ANSI C, Wydawnictwa Naukowo-Techniczne 1994*)) zintegrowane operatory przypisania (+=, *= itd.) definiowane były odwrotnie (tj. =+, =* itd.). Znakomicie utrudniało to interpretację wyrażeń takich jak:

```
x=-y
```

co mogłoby znaczyć

```
x = x - y
```

lub

```
x = (-y)
```

Ritchie szybko zauważył dwuznaczność takiego zapisu i zmodyfikował go do postaci znanej dzisiaj (+=, *= itd.). Mimo to wciąż stosowanych jest wiele odmian będących rodzajem wypośrodkowania między pierwotną wersją języka C Kernighana i Ritchie'ego a językiem ANSI C. Różnice między nimi dotyczą przede wszystkim:

- wprowadzenia prototypów funkcji i zmiany preambuły definicji funkcji, aby dostosować ją do stylu prototypów,
- wprowadzenia znaku wielokropka (...) do oznaczenia list argumentów o zmiennej długości,
- wprowadzenia słowa kluczowego `void` (dla funkcji, które nie zwracają wartości) i typu `void *` dla ogólnych zmiennych wskaźnikowych,
- wprowadzenie w preprocesorze mechanizmów scalania ciągów, wklejania elementu (*token-pasting*) i zamiany na ciąg (*string-izing*),
- dodanie w preprocesorze translacji „trygrafów" (*trigraph*) — trójznakowych sekwencji reprezentujących znaki specjalne,
- dodanie w preprocesorze dyrektywy `#pragma` i formalizacja pseudofunkcji `declared()`,

- wprowadzenie ciągów i znaków wielobajtowych, zapewniających obsługę języków narodowych,
- wprowadzenie słowa kluczowego `signed` (jako uzupełnienie słowa `unsigned`, stosowane w deklaracjach liczb całkowitych) i jednoargumentowego operatora plus (+).

Klasyfikowanie języka C

Szerokie możliwości języka C, dopuszczenie bezpośredniego operowania na adresach i danych w pamięci oraz strukturalne podejście do programowania sprawiają, że język ten klasyfikuje się jako „język programowania średniego poziomu". Znajduje to wyraz w mniejszej liczbie gotowych rozwiązań niż w językach wysokiego poziomu, takich jak BASIC, ale wyższym poziomie strukturalnym niż niskopoziomowy Assembler.

Słowa kluczowe

Pierwotna edycja języka C definiuje 27 słów kluczowych. Komitet ANSI dodał do nich 5 nowych. Wynikiem są dwa standardy języka, choć norma ANSI przejęła większość elementów od Kerninghana i Ritchie'ego. Oto lista:

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

Warto zwrócić uwagę, że niektóre kompilatory C wprowadzają dodatkowe słowa kluczowe, specyficzne dla środowiska sprzętowego. Warto zapoznać się z nimi.

Struktura języka C

Język C wymaga programowania strukturalnego. Oznacza to, że na program składa się pewna grupa nawzajem wywołujących się bloków kodu. Dostępne są różnorodne polecenia służące do konstruowania pętli i sprawdzania warunków:

```
do-while, for, while, if, case
```

Blok programu w języku C ujmowany jest w nawiasy klamrowe ({}). Może on być kompletną procedurą, nazywaną *funkcją* lub częścią kodu funkcji. Przyjrzyjmy się przykładowi:

```
if (x < 10)
{
  a = 1;
  b = 0;
}
```

Instrukcje wewnątrz nawiasów klamrowych wykonane zostaną tylko wtedy, gdy spełniony zostanie warunek `x < 10`.

Jako kolejny przykład przedstawimy pełny blok kodu funkcji, zawierający wewnątrz blok pętli:

```
int GET_X()
{
  int  x;

  do
  {
    printf ("\nWprowadz liczbe z zakresu od 0 do 10 ");
    scanf("%d",&x);
  }
  while(x < 0 || x > 10);
  return(x);
}
```

Zwróćmy uwagę, że każdy wiersz instrukcji zakończony jest średnikiem, o ile nie jest sygnałem początku bloku kodu (w takim przypadku kolejnym znakiem jest nawias klamrowy). Język C rozpoznaje wielkość liter, ale nie bierze pod uwagę białych znaków. Odstępy między poleceniami są pomijane, stąd konieczność użycia średnika, aby oznaczyć koniec wiersza. Tego rodzaju podejście powoduje, że następujące polecenia interpretowane są jako identyczne:

```
x = 0;
x        =0;
x=0;
```

Ogólna postać programu w języku C jest następująca:

- instrukcje preprocesora kompilacji,
- globalne deklaracje danych.
- deklaracje i definicje funkcji (włączając w to zawartość programu):

```
typ-zwracany main (lista parametrów)
{
  instrukcje
}
typ-zwracany f1 (lista parametrów)
{
  instrukcje
}
typ-zwracany f2 (lista parametrów)
{
  instrukcje
}
.
```

```
typ-zwracany fn (lista parametrów)
{
  instrukcje
}
```

Komentarze

Podobnie jak większość języków, C pozwala umieszczać w kodzie programu komentarze. Ich ogranicznikami są symbole /* i */:

```
/* To jest wiersz komentarza w języku C */
```

(Równie często korzysta się z komentarzy jednoliniowych, otrzymywanych poprzez sekwencję //, np.:

```
//To też jest wiersz komentarza  P.B.)
```

Biblioteki

Programy w języku C kompiluje się i łączy z funkcjami bibliotecznymi, dostarczanymi wraz z kompilatorem. Na biblioteki składają się funkcje standardowe, których działanie zdefiniowane zostało w normie ANSI. Ich powiązanie z konkretnym kompilatorem zapewnia dostosowanie do platformy sprzętowej. Wynika stąd, że standardowa funkcja biblioteczna `printf()` działa tak samo w systemach DEC VAX i IBM PC, choć różni się jej, zapisany w bibliotece, kod maszynowy. Programista C nie musi zagłębiać się w zawartość bibliotek, wymagana jest jedynie umiejętność ich stosowania i znajomość działania funkcji, które pozostają niezmienne na każdym komputerze.

Tworzenie programów

Kompilacja

Zanim zajmiemy się funkcjami, poleceniami, sekwencjami i innymi zaawansowanymi zagadnieniami, przyjrzyjmy się praktycznemu przykładowi, w którym doprowadzimy do skompilowania kodu. Kompilowanie programów C jest stosunkowo prostą czynnością, jednak różni się zależnie od stosowanego kompilatora. Kompilatory wyposażone w menu umożliwią skompilowanie, skonsolidowanie i uruchomienie programu jednym wciśnięciem klawisza. Podchodząc jednak do zagadnienia możliwie uniwersalnie i tradycyjnie, przeprowadzimy poniżej całą procedurę w oparciu o wiersz poleceń.

W dowolnym edytorze wprowadzamy poniższy fragment kodu i zapisujemy plik jako *przyklad.c*:

```
/*
  przykładowy komunikat tekstowy
*/
```

```
#include<stdio.h>
void main()
{
  printf( "Hello!\n" );
}
```

Kolejnym krokiem jest skompilowanie kodu do postaci pliku programu — dopiero wtedy można będzie go uruchomić (czy też wykonać). W wierszu poleceń w tym samym katalogu, w którym zapisaliśmy plik *przyklad.c*, wprowadzamy następujące polecenie kompilacji:

```
cc przyklad.c
```

Nie wolno zapominać, że składnia polecenia kompilacji zależy od kompilatora. Nasz przykład opiera się na standardzie języka C. Współcześnie jednak popularne jest stosowanie składni wywodzącej się z kompilatora GNU C:

```
gcc przyklad.c
```

Po wykonaniu takiego polecenia nasz kod jest już skompilowany i ma postać pliku programu, który możemy uruchomić. Wynik jego działania łatwo wydedukować z prostego kodu:

```
Hello!
Press any key to continue
```

To wszystko! Kompilowanie małych programów w C nie jest trudne, należy jedynie mieć świadomość szkodliwych niekiedy efektów ich działania. Programy przedstawiane na stronach tej książki i załączone na CD-ROM-ie są oczywiście znacznie bardziej skomplikowane, jednak zasady pozostają te same.

Typy danych

W języku C wyróżnia się cztery podstawowe typy danych: znakowy, całkowity, zmiennoprzecinkowy i nieokreślony. Odpowiadają im słowa kluczowe: `char`, `int`, `float` i `void`. Dalsze typy danych tworzy się na tej podstawie, dodając modyfikatory: `signed` (ze znakiem), `unsigned` (bez znaku), `long` (długa) i `short` (krótka). Modyfikator `signed` jest elementem domyślnym, co sprawia, że jego użycie może się okazać konieczne jedynie w wypadku gdy zastosowano przełącznik kompilacji nakazujący domyślne korzystanie ze zmiennych bez znaku. Rozmiar każdego typu danych zależy od platformy sprzętowej, jednak norma ANSI wyznacza pewne zakresy minimalne, zestawione w tabeli 6.1.

W praktyce tak określone konwencje oznaczają, że typ danych `char` nadaje się najlepiej do przechowywania zmiennych typu znacznikowego, takich jako kody stanu, o ograniczonym zakresie wartości. Można również korzystać z typu `int`. Gdy jednak zakres wartości nie przekracza 127 (lub 255 dla `unsigned char`), każda deklarowana w ten sposób zmienna przyczynia się do niepotrzebnego obciążania pamięci.

Natomiast trudniejsze jest pytanie o to, z którego typu liczb rzeczywistych korzystać — `float`, `double` czy `long double`. Gdy wymagana jest dokładność, na przykład w aplikacji stosowanej w księgowości, instynktownie powinniśmy użyć typu `long double`,

Tabela 6.1. *Rozmiary i zakresy typów danych języka C*

Typ	Rozmiar	Zakres
`char`	8	–128 do 127
`unsigned char`	8	0 do 255
`int`	16	–32 768 do 32 767
`unsigned int`	16	0 do 65 535
`long int`	32	–2 147 483 648 do 2 147 483 647
`unsigned long int`	32	0 do 4 294 967 295
`float`	32	precyzja 6-cyfrowa
`double`	64	precyzja 10-cyfrowa
`long double`	80	precyzja 10-cyfrowa

wiąże się to jednak z wykorzystaniem przez każdą zmienną 10 bajtów. Obliczenia na liczbach rzeczywistych nie są tak dokładne jak na liczbach całkowitych, warto więc zawsze rozważyć użycie typu `int` i „obejście" problemu. Typ danych `float` nie jest zbyt dobry, gdyż jego 6-cyfrowa precyzja nie zapewnia dokładności, na której zawsze będziemy mogli polegać. Ogólną zasadą jest korzystanie z typów całkowitych tak szeroko, jak tylko jest to możliwe, a gdy pojawia się konieczność użycia liczb rzeczywistych, wprowadzenie typu `double`.

Deklarowanie zmiennej

Każda zmienna musi zostać zadeklarowana przed użyciem. Ogólną postacią deklaracji zmiennej jest:

```
typ nazwa;
```

Aby więc przykładowo zadeklarować zmienną `x` typu `int`, przeznaczoną do przechowywania wartości z zakresu od –32 768 do 32 767, użyjemy instrukcji:

```
int x;
```

Ciągi znakowe deklarować można jako tabele znaków:

```
char nazwa[liczba_elementów];
```

Deklaracja ciągu o nazwie `nazwisko` i długości 30 znaków, wyglądać będzie następująco:

```
char nazwisko[30];
```

Tablice danych innych typów mogą mieć więcej niż jeden wymiar. Oto deklaracja dwuwymiarowej tablicy liczb całkowitych:

```
int x[10][10];
```

Elementy tablicy wywołujemy jako:

```
x[0][0]
x[0][1]
x[n][n]
```

Wyróżnia się trzy poziomy dostępu do zmiennych: lokalny, na poziomie modułu i globalny. Zmienna deklarowana wewnątrz bloku kodu będzie dostępna wyłącznie dla instrukcji wewnątrz tego bloku. Zmienna deklarowana poza blokami kodu funkcji, ale poprzedzona modyfikatorem `static`, będzie dostępna wyłącznie instrukcjom wewnątrz modułu kodu źródłowego. Zmienna deklarowana poza blokami kodu funkcji i niepoprzedzona modyfikatorem będzie dostępna dla dowolnych instrukcji w dowolnym module programu. Na przykład:

```
int blad;
static int a;

void main() (Co prawda funkcja main działa i bez deklaracji wartości zwracanej,
jednak w takim przypadku wyświetla się ostrzeżenie (bo kompilator domyślnie
przyjmuje ją jako int i szuka funkcji return. Aby tego uniknąć, w każdym następnym
przykładzie dopisuję void P.B.)
{
  int x;
  int y;
}

funkcjaa()
{
  /* Sprawdzenie czy zmienna a jest równa 0 */
  if (a == 0)
  {
    int b;
    for(b = 0; b < 20; b++)
      printf ("\nHello World");
  }

}
```

W powyższym przykładzie zmienna `blad` jest dostępna dla wszystkich, kompilowanych jako jeden program, modułów kodu źródłowego. Zmienna `a` jest osiągalna dla wszystkich instrukcji w funkcjach `main()` i `funkcjaa()`, ale pozostaje niewidoczna z poziomu innych modułów. Zmienne `x` i `y` są dostępne wyłącznie instrukcjom wewnątrz funkcji `main()`. Z kolei zmienna `b` może być użyta wyłącznie przez instrukcje wewnątrz bloku kodu po instrukcji `if`.

Jeżeli drugi blok kodu faktycznie ma skorzystać ze zmiennej `blad`, wymagane będzie umieszczenie w nim deklaracji zmiennej globalnej `extern`:

```
extern int blad;

funkcjab()
{
}
```

Język C nie stawia szczególnych przeszkód w przypisywaniu do siebie różnych typów danych. Przykładowo możemy zadeklarować zmienną typu `char`, co spowoduje przypisanie do przechowywania jej wartości jednego bajtu danych. Można podjąć próbę przypisania do niej wartości spoza tego zakresu:

```
void main()
{

x = 5000;

}
```

Zmienna `x` może przechowywać wartości z zakresu od –127 do 128, a więc wartość 5000 nie zostanie przypisana. `x` przyjmie jednak wartość 136.

Potrzeba przypisania różnych typów danych nie jest niczym oryginalnym. Aby powstrzymać kompilator od generowania ostrzeżeń o takich operacjach, można skorzystać z *instrukcji konwersji* (*cast statement*), informując kompilator o tym, że operacja wykonywana jest świadomie. Instrukcję taką budujemy, umieszczając przed zmienną lub wyrażeniem nazwę typu danych ujętą w nawiasy:

```
void main()
{
  float x;
  int y;

  x = 100 / 25;

  y = (int)x;
}
```

Operacja rzutowania `(int)` informuje kompilator o konieczności konwersji wartości zmiennej zmiennoprzecinkowej `x` do liczby całkowitej, zanim ta zostanie przypisana do zmiennej `y`.

Parametry formalne

Funkcja w języku C może przyjmować parametry przekazywane przez funkcję wywołującą. Parametry te deklaruje się podobnie jak zmienne, podając ich nazwy wewnątrz towarzyszących nazwie funkcji nawiasów:

```
int MNOZ(int x, int y)
{
  /* Zwróć parametr x pomnożony przez parametr y */
  return(x * y);
}

void main()
{
  int a;
  int b;
  int c;

  a = 5;
  b = 7;
  c = MNOZ(a,b);

  printf("%d razy %d równa się %d\n",a,b,c);
}
```

Modyfikatory dostępu

Stosuje się dwa modyfikatory dostępu: `const` i `volatile`. Wartość zmiennej zadeklarowanej jako `const` nie może zostać zmieniona przez program, wartość zmiennej zadeklarowanej jako `volatile` może zostać zmieniona przez program. Dodatkowo, zadeklarowanie zmiennej jako `volatile` uniemożliwia kompilatorowi zaalokowanie jej do rejestru i ogranicza przeprowadzaną na niej optymalizację.

Typy klas przechowywania zmiennych

Język C przewiduje cztery rodzaje przechowywania zmiennych: `extern`, `static`, `auto` i `register`. Typ `extern` umożliwia modułowi kodu źródłowego dostęp do zmiennej zadeklarowanej w innym module. Zmienne `static` dostępne są wyłącznie z poziomu bloku kodu, w którym zostały zadeklarowane. Dodatkowo, jeżeli zmienna ma zasięg lokalny, zachowuje swoją wartość między kolejnymi wywołaniami bloku kodu.

Zmienne rejestrowe (`register`) są, gdy tylko jest to możliwe, przechowywane w rejestrach procesora. Zapewnia to najszybszy dostęp do ich wartości. Typ `auto` stosuje się wyłącznie w odniesieniu do zmiennych lokalnych. Nakazuje on zachowywanie wartości zmiennej lokalnej. Ponieważ jest to modyfikator domyślny, rzadko można spotkać go w programach.

Operatory

Operatory to elementy kodu, które nakazują wykonanie obliczeń na zmiennych. W języku C dostępne są następujące:

`&`	adres,
`*`	pośredniość,
`+`	plus jednoargumentowy,
`-`	minus jednoargumentowy,
`~`	dopełnienie bitowe,
`!`	negacja logiczna,
`++`	jako prefiks — preinkrementacja, jako sufiks — postinkrementacja,
`--`	jako prefiks — predekrementacja, jako sufiks — postdekrementacja,
`+`	dodawanie,
`-`	odejmowanie,
`*`	mnożenie,
`/`	dzielenie,
`%`	reszta z dzielenia (modulo),
`<<`	przesunięcie w lewo,
`>>`	przesunięcie w prawo,
`&`	bitowa operacja AND,

`|`	bitowa operacja OR,
`^`	bitowa operacja XOR,
`&&`	logiczna operacja AND,
`||`	logiczna operacja OR,
`=`	przypisanie,
`*=`	przypisanie iloczynu,
`/=`	przypisanie ilorazu,
`%=`	przypisanie reszty (modułu),
`+=`	przypisanie sumy,
`-=`	przypisanie różnicy,
`<<=`	przypisanie przesunięcia w lewo,
`>>=`	przypisanie przesunięcia w prawo,
`&=`	przypisanie wyniku bitowej operacji AND,
`|=`	przypisanie wyniku bitowej operacji OR,
`^=`	przypisanie wyniku bitowej operacji XOR,
`<`	mniejsze niż,
`>`	większe niż,
`<=`	mniejsze lub równe,
`>=`	większe lub równe,
`==`	równe,
`!=`	różne od,
`.`	bezpośredni selektor składnika,
`->`	pośredni selektor składnika,
`a ? x : y`	jeżeli `a` to prawda, to `x`, w przeciwnym razie `y`,
`[ ]`	definiowanie tablic,
`()`	nawiasy oddzielają warunki i wyrażenia,
`...`	wielokropek wykorzystuje się w listach parametrów formalnych prototypów funkcji do deklarowania zmiennej liczby parametrów lub parametrów zmiennych typów.

Aby zilustrować sposób korzystania z podstawowych operatorów, przyjrzyjmy się krótkiemu programowi:

```
void main()
{
  int a;
  int b;
  int c;
  a = 5;   /*Przypisanie zmiennej a wartości 5*/
  b = a/2; /*Przypisanie zmiennej b wartości a podzielonej przez 2*/
  c = b*2;  /*Przypisanie zmiennej c wartości b pomnożonej przez 2*/

  if (a == c) /*Sprawdzenie czy a ma taką samą wartość jak c*/
```

```
    puts("Zmienna a jest parzysta");
  else
    puts("Zmienna a jest nieparzysta");
}
```

Typowym sposobem zwiększenia wartości zmiennej o 1 jest wiersz:

```
x = x + 1
```

Język C dostarcza operatora inkrementacji, wystarczy więc napisać:

```
x++
```

W podobny sposób korzystamy z operatora dekrementacji, czyli zmniejszania wartości o 1:

```
x--
```

Pozostałe operatory matematyczne wykorzystujemy podobnie. Warto jednak pamiętać o wprowadzanych przez język C możliwościach zapisu skróconego:

Zapis typowy	Zapis w języku C
`x = x + 1`	`x++`
`x = x - 1`	`x--`
`x = x * 2`	`x *= 2`
`x = x / y`	`x /= y`
`x = x % 5`	`x %= 5`

Funkcje

Funkcje to procedury kodu źródłowego tworzące program w języku C. Ich ogólną postacią jest:

```
zwracany_typ nazwa_funkcji(lista_parametrów)
{
  instrukcje
}
```

Zwracany_typ to typ zwracanej przez funkcję wartości: `char`, `int`, `double`, `void` itp. Kod wewnątrz funkcji C pozostaje niewidoczny dla innych funkcji C. Nie można wykonywać skoków z jednej funkcji do wnętrza innej. Funkcje mogą jedynie wywoływać inne funkcje. Nie wolno również definiować funkcji wewnątrz innych funkcji. Definicja musi zostać umieszczona bezpośrednio na poziomie modułu kodu.

Parametry przekazywane są do funkcji jako wartości lub jako odwołania (wskaźniki). Gdy parametr jest przekazywany jako wartość, funkcja otrzymuje kopię tej wartości. Parametr przekazywany jako odwołanie jest jedynie wskaźnikiem do właściwego parametru. Pozwala to na zmianę jego wartości z poziomu wywołanej funkcji. W poniższym przykładzie przekazujemy dwa parametry jako wartość do funkcji `funkcjaa()`,

która następnie podejmuje próbę zmiany wartości przekazanych zmiennych. Drugim krokiem jest przekazanie tych samych parametrów do funkcji `funkcjab()`, która również podejmuje próbę zmiany wartości zmiennych:

```
#include <stdio.h>

int funkcjaa(int x, int y)
{
  /* Funkcja przyjmuje dwa parametry jako wartości, x i y */

  x = x * 2;
  y = y * 2;

  printf ("\nWartość x w funkcjaa() %d. Wartość y w funkcjaa() %d",x,y);

  return(x);
}


int funkcjab(int *x, int *y)
{
  /* Funkcja przyjmuje dwa parametry jako odwołania, x i y */

  *x = *x * 2;
  *y = *y * 2;

  printf ("\nWartość x w funkcjab() %d. Wartość y w funkcjab() %d",*x,*y);

  return(*x);
}

void main()
{
  int x;
  int y;
  int z;

  x = 5;
  y = 7;

  z = funkcjaa(x,y);
  z = funkcjab(&x,&y);

  printf ("\nWartość x %d, wartość y %d, wartość z %d",x,y,z);
}
```

`funkcjab()` nie zmienia wartości otrzymanych parametrów. Modyfikowana jest zawartość wskazywanych parametrami adresów pamięci. O ile `funkcjaa()` otrzymuje z funkcji `main()` wartości zmiennych `x` i `y`, `funkcjab()` otrzymuje z funkcji `main()` ich adresy w pamięci.

Przekazywanie tablicy do funkcji

Następujący program przekazuje do funkcji tablicę, a funkcja nadaje wartości elementom tablicy:

```
#include <stdio.h>

void funkcjaa(int x[])
{
  int n;

  for(n = 0; n < 100; n++)
  x[n] = n;
}

void main()
{
  int tablica[100];
  int licznik;

  funkcjaa(tablica);

  for(licznik = 0; licznik < 100; licznik++)
    printf ("\nWartość elementu %d wynosi %d",licznik, tablica[licznik]);
}
```

Parametr funkcji, `int x[]`, jest tablicą dowolnej długości. Deklaracja taka jest możliwa, ponieważ kompilator przekazuje jedynie adres początkowy tablicy, a nie wartości poszczególnych jej elementów. Konsekwencją tego jest fakt, że funkcja może zmieniać wartości elementów tablicy. Aby uniemożliwić funkcji wprowadzanie modyfikacji, konieczne jest użycie typu `const`:

```
funkcjaa(const int x[])
{
}
```

Przy takiej deklaracji wiersz zmieniający zawartość tablicy wywołałby błąd kompilacji. Określenie parametru jako wartości stałej nie likwiduje jednak pośredniości jego przekazania. Ilustruje to poniższy program:

```
#include <stdio.h>

 void funkcjaa(const int x[])
{
  int *ptr;
  int n;

/*Ten wiersz generuje ostrzeżenie 'suspicious pointer conversion'*/
/*(niebezpieczna konwersja wskaźnika)*/
/*x jest wskaźnikiem const, a ptr - nie*/
  ptr = x;

  for(n = 0; n < 100; n++)
  {
    *ptr = n;
    ptr++;
  }
}

void main()
{
```

```
  int tablica[100];
  int licznik;

  funkcjaa(tablica);

  for(licznik = 0; licznik < 100; licznik++)
    printf("\nWartość elementu %d wynosi %d",licznik,tablica[licznik]);
}
```

Przekazywanie parametrów funkcji main()

Język C umożliwia przekazanie parametrów do uruchamianego programu z poziomu systemu operacyjnego. Do ich odczytania wykorzystuje się zmienne `argc` i `argv[]`:

```
#include <stdio.h>

void main(int argc, char *argv[])
{
  int n;

  for(n = 0; n < argc; n++)
  printf ("\nWartosc parametru %d to %s",n,argv[n]);
}
```

Parametr `argc` przechowuje liczbę przekazanych programowi parametrów. W tablicy `argv[]` zapisane są ich adresy; `argv[0]` jest zawsze nazwą uruchamianego programu. Mechanizm ten ma szczególne znaczenie dla aplikacji wymagających dostępu do plików systemowych i danych. Rozważmy następującą sytuację: mała aplikacja obsługi baz danych przechowuje swoje dane w pojedynczym pliku *dane.dat*; aplikacja ta musi zostać tak zaprojektowana, aby można było uruchomić ją z dowolnego katalogu, czy to na dysku twardym, czy dyskietce; musi również zapewnić uruchamianie za pośrednictwem ścieżki wyszukiwania DOS-u (`path`). Do poprawnej pracy aplikacji jest więc wymagane, aby zawsze mogła odnaleźć plik *dane.dat*. Rozwiązanie takie zapewni przyjęcie założenia, że plik danych jest zawsze w identycznym katalogu co sam program. Poniższy fragment ilustruje wykorzystanie parametrów `argc` i `argv` w celu utworzenia ścieżki do pliku danych aplikacji:

```
#include <string.h>

char nazwa_pliku[160];

void main(int argc, char *argv[])
{
  char *plik_danych = "DATA.DAT";
  char *p;

  strcpy(nazwa_pliku,argv[0]);

p = strstr(nazwa_pliku, ".exe"); (kompilator tworzy plik z rozszerzeniem o małych
➥ literach P.B.)
  if (p == NULL)
  {
    /* Plik uruchomieniowy jest plikiem .COM */
```

```
p = strstr(nazwa_pliku, ".com");
  }

  /* Wyszukujemy ostatni ukośnik */
  while(*(p-1) != '\\')
    p--;

  strcpy(p,plik_danych);
}
```

Przedstawiony program tworzy i zapisuje w zmiennej `nazwa_pliku` ciąg postaci *ścieżka\dane.dat*. Jeżeli więc przykładową nazwą pliku uruchomieniowego będzie *test.exe* i zostanie on umieszczony w katalogu *\borlandc*, zmiennej `nazwa_pliku` przypisany zostanie ciąg `\borlandc\dane.dat`.

Wyjście z funkcji

Polecenie `return` powoduje natychmiastowe wyjście z funkcji. Jeżeli w deklaracji funkcji podano typ zwracanej wartości, w poleceniu `return` należy użyć parametru tego samego typu.

Prototypy funkcji

Prototypy funkcji umożliwiają kompilatorowi C sprawdzanie poprawności przekazywanych, do i z funkcji, danych. Ma to istotne znaczenie jako zabezpieczenie przed przekroczeniem zakresu zaalokowanego dla zmiennej obszaru pamięci. Prototyp funkcji umieszcza się na początku programu po poleceniach preprocesora (takich jak `#include`) i przed deklaracjami funkcji.

Polecenia preprocesora C

W języku C w treści kodu źródłowego można umieszczać polecenia dla kompilatora. Określa się je terminem *polecenia preprocesora*. Norma ANSI definiuje następujące:

```
#if
#ifdef
#ifndef
#else
#elif
#endif
#include
#define
#undef
#line
#error
#pragma
```

Wszystkie polecenia preprocesora rozpoczyna znak krzyżyka (hash), czyli #. Każde wymaga osobnego wiersza kodu (uzupełnionego ewentualnie komentarzem). Poniżej przedstawiamy krótkie omówienie.

#define

Polecenie #define tworzy identyfikator, który kompilator zastąpi podanym ciągiem w danym module kodu źródłowego. Na przykład:

```
#define FALSE 0
#define TRUE !FALSE
```

Kompilator zastąpi wszystkie dalsze wystąpienia ciągu FALSE znakiem 0, a wszystkie dalsze wystąpienia ciągu TRUE — ciągiem !0. Zastępowaniu *nie* podlegają identyfikatory wewnątrz znaków cudzysłowu, a więc wiersz:

```
printf ("TRUE");
```

nie zostanie zmieniony, ale

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

podlega modyfikacji.

Polecenie #define może również zostać użyte do definiowania makr, także makr z parametrami. Do zapewnienia poprawności zastąpień zaleca się ujmowanie parametrów w nawiasy. W poniższym przykładzie deklarujemy makro o nazwie larger(), przyjmujące dwa parametry i zwracające ten z nich, którego wartość jest większa.

```
#include <stdio.h>

#define larger(a,b) (a > b) ? (a) : (b)

int main()
{
  printf("\n%d jest większe",larger(5,7));
}
```

#error

Polecenie #error powoduje przerwanie procesu kompilacji i wyświetlenie podanego tekstu, na przykład:

```
#error SKOMPILOWANE DO MODULU B
```

powoduje zatrzymanie kompilacji i wyświetlenie:

```
SKOMPILOWANE DO MODULU B
```

#include

Polecenie #include nakazuje kompilatorowi odczytanie i przetworzenie zawartości dodatkowego pliku źródłowego. Nazwa pliku musi zostać ujęta w cudzysłów lub wstawiona między znaki <>, na przykład:

```
#include "module2.c"
#include <stdio.h>
```

Jeżeli nazwa pliku została wpisana między znaki <>, kompilator wyszukuje go w katalogu określonym w konfiguracji. Jest to zasada ogólna.

#if, #else, #elif, #endif

Grupa poleceń #if dostarcza mechanizmu kompilacji warunkowej. Stosowana jest dość typowa składnia:

```
#if wyrażenie_stale
  instrukcje
#else
  instrukcje
#endif
```

Polecenie #elif to skrócona postać #else if:

```
#if wyrażenie
  instrukcje
#elif wyrażenie
  instrukcje
#endif
```

#ifdef, #ifndef

Rozwinięciem tych poleceń jest #if defined (jeżeli zdefiniowano) i #if not defined (jeżeli nie zdefiniowano). Konstrukcje składniowe są następujące:

```
#ifdef nazwa_makra
  instrukcje
#else
  instrukcje
#endif

#ifndef nazwa_makra
  instrukcje
#else
  instrukcje
#endif
```

nazwa_makra to identyfikator utworzony za pomocą deklaracji #define.

#undef

Polecenie #undef usuwa definicję makra utworzonego przy użyciu wcześniejszej instrukcji #define.

#line

Polecenie #line modyfikuje zmienne globalne kompilatora __LINE__ i __FILE__. Ogólną postacią instrukcji jest:

```
#line numer "nazwa_pliku"
```

Wartość *numer* zostaje umieszczona w zmiennej __LINE__, a "*nazwa_pliku*" — w zmiennej __FILE__.

#pragma

Umożliwia korzystanie z poleceń specyficznych dla kompilatora.

Instrukcje sterujące

Jak w każdym języku programowania, również w C, znajdziemy instrukcje sprawdzające wartość wyrażenia. Wynikiem takiego sprawdzenia jest wartość `TRUE` lub `FALSE`. Wartości `FALSE` odpowiada liczba `0`, a `TRUE` — liczba różna od zera.

Instrukcje wykonania warunkowego

Podstawową instrukcją wykonania warunkowego jest `if` o następującej składni:

```
if (wyrażenie)
   instrukcje
else
   instrukcje
```

gdzie *instrukcje* może być instrukcją pojedynczą lub ujętym w nawiasy klamrowe blokiem kodu. Element `else` jest opcjonalny. Jeżeli wartością *wyrażenie* jest `TRUE`, wykonywana jest instrukcja podana bezpośrednio po nim. W pozostałych przypadkach wykonywana jest instrukcja podana po słowie `else` (o ile ta część składni została użyta).

Alternatywą dla konstrukcji `if...else` jest polecenie `?:` w postaci:

```
wyrażenie ? instrukcja_prawda : instrukcja_fałsz
```

Jeżeli wartością wyrażenia jest `TRUE`, wykonywana jest pierwsza instrukcja. W pozostałych przypadkach wykonywana jest instrukcja druga. Ilustruje to przykład:

```
#include <stdio.h>

void main()
{
  int x;
  x = 6;

  printf("\nx to liczba %s", x % 2 == 0 ? "parzysta" : "nieparzysta");
}
```

Język C oferuje również instrukcję `switch`, ułatwiającą porównywanie wyrażenia z pewną listą wartości. Wykonywane są instrukcje powiązane z pierwszą dopasowaną wartością listy. Składnia polecenia `switch` jest następująca:

```
switch (wyrażenie)
{
  case wartość1 :   instrukcje
    break;
  case wartość2 :   instrukcje
    break;
.
.
```

```
  case wartośćn :   instrukcje
    break;
  default :   instrukcje
}
```

Użycie instrukcji `break` nie jest wymagane, ale jej pominięcie powoduje dalsze porównywanie wyrażenia z kolejnymi elementami listy wartości.

```
#include <stdio.h>

void main()
{
  int x;

  x = 6;

  switch (x)
  {
    case 0 : printf ("\nx równa się zero");
      break;
    case 1 : printf ("\nx równa się jeden");
      break;
    case 2 : printf ("\nx równa się dwa");
      break;
    case 3 : printf ("\nx równa się trzy");
      break;
    default : printf ("\nx jest większe od trzech");
  }
}
```

Instrukcje `switch` można zagnieżdżać.

Instrukcje iteracji

W języku C stosuje się trzy instrukcje pętli (iteracji): `for`, `while` i `do-while`. Składnia pętli `for` jest następująca:

```
for(inicjalizacja;warunek;inkrement)
   instrukcje
```

Jest ona szczególnie przydatna, gdy korzystamy z licznika, jak w poniższym przykładzie wyświetlającym zestaw znaków ASCII:

```
#include <stdio.h>

void main()
{
  int x;

  for(x = 32; x < 128; x++)
    printf ("%d\t%c\t",x,x);
}
```

Dopuszczalna jest również nieskończona pętla `for`:

```
for(;;)
{
  instrukcje
}
```

Język C pozwala używać też pustych instrukcji. Poniższa pętla usuwa z ciągu początkowe znaki odstępu:

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

Warto zwrócić uwagę na średniki odpowiadające inicjalizacji pętli i pustej instrukcji.

Pętla `while` ma konstrukcję nieco prostszą:

```
while(warunek)
   instrukcje
```

Instrukcja lub blok instrukcji (ujęty w nawiasy klamrowe) będą powtarzane do czasu, gdy wyrażenie warunku przyjmie wartość `FALSE`. Jeżeli wyrażenie nie jest prawdziwe jeszcze przed wejściem do pętli, instrukcje nie będą wykonywane w ogóle. Jest to istotna różnica w stosunku do pętli `do-while`, która zawsze zostaje wykonana co najmniej raz. Jej składnia to:

```
do
{
  instrukcje
}
while(warunek);
```

Instrukcje skoku

Instrukcja `return` pozwala powrócić z funkcji wykonywanej do funkcji, z której ta została wywołana. W zależności od zadeklarowanego typu wartości zwracanej przez funkcję instrukcja `return` może wymagać odpowiedniego parametru:

```
int MULT(int x, int y)
{
  return(x * y);
}
```

lub

```
void funkcjaa()
{
  printf ("\nHello World");
  return; (w tym wypadku return nie jest konieczny P.B.)
}
```

Instrukcja `break` służy do wychodzenia z pętli lub instrukcji `switch`. W przypadku pętli powoduje to jej przedwczesne zakończenie, jak w poniższym przykładzie:

```
#include <stdio.h>

void main()
{
  int x;
```

```
  for(x = 0; x < 256; x++)
  {
    if (x == 100)
      break;

    printf ("%d\t",x);
  }
}
```

Uzupełnieniem `break` jest polecenie `continue`, wymuszające przeprowadzenie następnej iteracji pętli. Kolejną wykonywaną instrukcją jest w tym przypadku instrukcja pętli (dalsze instrukcje w iterowanym bloku są pomijane). Dostępna jest również funkcja przedwczesnego zakończenia wykonywania programu — `exit()`. Można za jej pomocą przekazać wartość zwracaną do programu wywołującego:

```
exit(wartość_zwracana);
```

Continue

Słowo kluczowe `continue` nakazuje skok do instrukcji kontrolnej pętli. W przypadku pętli zagnieżdżonych jest to instrukcja pętli wewnętrznej (`while`, `do...while()`). To sposób na łagodne zakończenie pętli jak w poniższym przykładzie, gdzie odczytujemy zapisane w pliku ciągi:

```
#include <stdio.h>

void main()
{
  FILE *fp;
  char *p;
  char buff[100];

  fp = fopen("dane.txt","r");
  if (fp == NULL)
  {
    fprintf(stderr,"Nie można otworzyć pliku data.txt");
    exit(0);
  }

  do
  {
    p = fgets(buff,100,fp);
    if (p == NULL)
      /* Wymuszenie wyjścia */
      continue;
    puts(p);
  }
  while(p);
}
```

W przypadku pętli `for` instrukcja `continue` powoduje najpierw wykonanie wyrażenia inkrementacji, a dopiero po nim następuje sprawdzenie warunku zakończenia.

Wejście-wyjście

Pobieranie danych

Program w języku C może pobierać dane z konsoli (która jest standardowym urządzeniem wejściowym), pliku lub portu. Ogólnym poleceniem odczytu danych ze standardowego strumienia wejściowego `stdin` jest `scanf()`. Skanuje ono po jednym znaku kolejne pola wejściowe. Podlegają one formatowaniu zgodnie z pierwszym z przekazanych funkcji `scanf()` parametrów. Następnie pole zostaje zapisane pod adresem przekazanym jako kolejny parametr wywołania funkcji. Przykładowy program odczytuje pojedynczą liczbę całkowitą ze strumienia `stdin`:

```
void main()
{
  int x;

  scanf("%d", &x);
}
```

Warto zwrócić uwagę na operator użyty jako prefiks zmiennej `x` na liście parametrów wywołania funkcji `scanf()`. Funkcja ta zapisuje bowiem wartość pod określonym *adresem*, nie posługując się mechanizmem przypisywania wartości zmiennej. Ciągiem formatującym jest ciąg znakowy, który może zawierać trzy typy danych: *znaki odstępu* (spacja, tabulator, przejście do nowego wiersza), *znaki właściwe* (wszystkie znaki ASCII z wyjątkiem znaku %) i *specyfikatory formatowania*. Specyfikatory te mają następującą składnię:

```
%[*][szerokość][h|l|L]typ
```

Oto przykład:

```
#include <stdio.h>

void main()
{
  char nazwisko[30];
  int wiek;

  printf ("Podaj nazwisko i wiek ");
  scanf("%30s%d",nazwisko,&wiek);
  printf ("\n%s %d",nazwisko,wiek);
}
```

Zwróćmy uwagę na wiersz `#include <stdio.h>` — nakazuje on kompilatorowi przetwarzanie pliku nagłówkowego *stdio.h*, w którym zawarte są prototypy funkcji `scanf()` i `printf()`. Po uruchomieniu tego prostego programu łatwo przekonamy się, że użycie znaku odstępu przerwie wprowadzanie pierwszego pola danych.

Alternatywną funkcją pobierania danych jest `gets()`, odczytująca ciąg znaków ze strumienia `stdin` do momentu napotkania znaku nowego wiersza. W ciągu docelowym znak nowego wiersza zastąpiony zostaje znakiem `NULL (0)`. Charakterystyczna dla tej funkcji jest możliwość odczytywania znaków odstępu. Oto nowa wersja powyższego programu (korzystająca z `gets()` w miejsce `scanf()`):

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

void main()
{
  char dane[80];
  char *p;
  char nazwisko[30];
  int wiek;

  printf ("\nPodaj nazwisko i wiek ");
  /* Odczyt ciągu danych */
  gets(dane);

  /* p jest wskaźnikiem do ostatniego znaku pobieranego ciągu */
  p = &dane[strlen(dane) - 1];

  /* Usuwamy spacje końcowe, zastępując je znakami NULL */
  while(*p == ' '){
    *p = 0;
    p--;
  }

  /* Lokalizujemy ostatnią spację w ciągu */
  p = strrchr(dane,' ');

  /* Odczytujemy wiek i zamieniamy na liczbę */
  wiek = atoi(p);

  /* Wstawiamy znak końca ciągu przed polem wieku */
  *p = 0;

  /* Kopiujemy ciąg danych do zmiennej */
  strcpy(nazwisko, dane);

  /* Wyświetlamy wyniki operacji */
  printf ("\nNazwisko: %s, wiek: %d", nazwisko, wiek);
}
```

Wyprowadzanie danych

Podstawową funkcją wyprowadzania danych jest `printf()`. Jest ona podobna do `scanf()` z tą różnicą, że zapisuje dane do standardowego strumienia wyjściowego `stdout`. Funkcja pobiera listę pól danych wyjściowych, odpowiednio stosuje specyfikatory formatowania i wyprowadza wynik. Można stosować takie same przekształcenia formatujące jak w przypadku funkcji `scanf()`, jak również dodatkowe znaczniki:

- `-` wyrównuje dane wyjściowe do lewej, uzupełniając je z prawej strony znakami odstępu międzywyrazowego (spacji),
- `+` wymusza poprzedzanie liczb znakiem.

Nieco odmienna jest także postać specyfikatora szerokości. Jest on rozbudowany o element określający precyzję:

szerokość.precyzja

Aby więc wyświetlić liczbę zmiennoprzecinkową z dokładnością do trzech miejsc dziesiętnych, piszemy:

```
printf ("%.3f",x);
```

Poniżej przedstawiamy listę specjalnych stałych znakowych, które mogą pojawić się na liście parametrów funkcji `printf()`:

`\n`	nowy wiersz (NL),
`\r`	powrót karetki (CR),
`\t`	tabulator,
`\b`	znak cofania (backspace),
`\f`	znak nowej strony,
`\v`	tabulator pionowy,
`\\`	ukośnik odwrotny (backslash),
`\'`	apostrof,
`\"`	cudzysłów,
`\?`	znak zapytania,
`\o`	ciąg w notacji ósemkowej,
`\x`	ciąg w notacji szesnastkowej.

Kolejny program ilustruje, w jaki sposób wyświetlić liczbę całkowitą w postaci dziesiętnej, szesnastkowej i ósemkowej. Liczba `04` po znaku procentów (`%`) w instrukcji `printf()` nakazuje kompilatorowi dopełnienie wyświetlanej liczby do szerokości co najmniej czterech cyfr:

```
/* Prosty program konwersji liczb dziesiętnych */
/* do postaci szesnastkowej i ósemkowej */

#include <stdio.h>

void main()
{
  int x;

  do
  {
    printf ("\nPodaj liczbę (lub 0, aby zakończyć) ");
    scanf("%d",&x);
    printf ("%04d %04X %04o",x,x,x);
  }
  while (x != 0);

}
```

Do funkcji pokrewnych `printf()` należy `fprintf()`, której prototyp ma postać:

```
fprintf(FILE *fp, char *format[,argument,...]);
```

Jej zadaniem jest przesyłanie sformatowanych danych wyjściowych do określonego strumienia plikowego.

Kolejną tego rodzaju funkcją jest `sprintf()` o prototypie:

```
sprintf(char *s, char *format[,argument,...]);
```

Alternatywą dla `printf()` jest `puts()`, funkcja przesyłająca prosty ciąg do strumienia `stdout`. Przesyłany ciąg zostaje automatycznie uzupełniony znakiem nowego wiersza. Jest to rozwiązanie szybsze od `printf()`, jednak jego możliwości są ograniczone.

Bezpośrednia wymiana danych z konsolą

Do przesyłania i odczytu danych z konsoli (klawiatury i ekranu) można wykorzystywać również bezpośrednie funkcje we-wy. Wyróżnia je litera „c" na początku — odpowiednikiem `printf()` jest więc `cprintf()`, a odpowiednikiem `puts()` — funkcja `cputs()`. Różnice między funkcjami bezpośredniej wymiany danych a funkcjami standardowymi są następujące.

- Nie są wykorzystywane strumienie predefiniowane, nie można więc przekierować danych przesyłanych funkcjami komunikacji bezpośredniej.
- Funkcji bezpośrednich nie można przenosić między różnymi systemami operacyjnymi (m.in. nie można z nich korzystać w programach dla Windows).
- Funkcje bezpośrednie są szybsze niż standardowe.
- Nie zapewniają współpracy ze wszystkimi trybami wyświetlania (zwłaszcza trybami graficznymi VESA).

Wskaźniki

Wskaźnik to zmienna, która przechowuje adres elementu danych w pamięci. Deklaracja wskaźnika jest podobna do deklaracji zwykłej zmiennej, ale nazwa poprzedzana jest znakiem gwiazdki (*), na przykład:

```
char *p;
```

Powyższy wiersz deklaruje zmienną `p` jako wskaźnik do zmiennej typu `char`.

Wykorzystanie wskaźników dostarcza szerokich możliwości, wymaga jednak szczególnej uwagi. Skutki przypisania błędnego adresu są najczęściej nieprzewidywalne. Oto przykład prostego programu, w którym wykorzystywany jest wskaźnik:

```
#include <stdio.h>

void main()
{
  int a;
  int *x;

  /* x jest wskaźnikiem do danych typu int */

  a = 100;
  x = &a;

  printf("\nZmienna a przechowuje wartość %d pod adresem %p.",a,x);
}
```

Wartości wskaźników można zwiększać i zmniejszać, dopuszczalne są również inne operacje matematyczne. Typowym zastosowaniem wskaźników jest zapewnienie dynamicznego przydziału pamięci. W trakcie pracy programu często pojawia się potrzeba przejściowego (tymczasowego) zaalokowania bloku pamięci. Korzystamy wówczas z funkcji `malloc()`:

```
wskaźnik_dowolnego_typu = malloc(liczba_bajtów);
```

Funkcja `malloc()` zwraca wskaźnik typu `void`, co oznacza, że może on wskazywać dane dowolnego typu — `int`, `char`, `float` itd. W poniższym przykładzie alokujemy pamięć dla tabeli 1000 liczb całkowitych.

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

void main()
{
  int *x;
  int n;

  /* x jest wskaźnikiem do danych typu int */

  /* Tworzymy tablicę 1000-elementową */
  /* sizeof() dostarcza kompilatorowi informacji o liczbie */
  /* bajtów wymaganej do przechowywania zmiennej typu int */

  x = malloc(1000 * sizeof(int));

  /* Sprawdzamy czy alokacja została wykonana */
  if (x == NULL)
  {
    printf("\nNie można zaalokować pamięci dla 1000-elementowej tablicy wartości
    ➥int");
    exit(0);
  }

  /* Przypisujemy wartości poszczególnym elementom */
  for(n = 0; n < 1000; n++)
  {
    *x = n;
    x++;
  }

  /* Przywracamy x wartość adresu początkowego tabeli */
  x -= 1000;

  /* Wyświetlamy wartości tabeli */
  for(n = 0; n < 1000; n++){
    printf("\nElement %d przechowuje wartość %d",n,*x);
    x++;
  }
  /* Po użyciu, dealokujemy blok pamięci */
  free(x);
}
```

Wskaźniki wykorzystuje się również w odniesieniu do tablic znaków, czyli *ciągów* (*strings*). Ponieważ wszystkie ciągi w programach C kończy bajt o wartości 0, korzystając ze wskaźnika, możemy policzyć znaki w ciągu:

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

void main()
{
  char *p;
  char tekst[100];
  int dlugosc;

  /* Inicjujemy zmienną 'tekst' */
  strcpy(tekst,"To jest ciag znakowy");

  /* Ustawiamy wartość zmiennej p na początek tekstu */
  p = tekst;

  /* Inicjujemy zmienną długość */
  dlugosc = 0;

  /* Zliczamy znaki w zmiennej tekst */
  while(*p)
  {
    dlugosc++;
    p++;
  }

  /* Wyświetlamy wynik */
   printf("\nDlugosc ciagu znakowego to: %d",dlugosc);
}
```

Wymaganą do zaadresowania 1 MB pamięci 20-bitową liczbę dzieli się na dwie wartości: *przesunięcie* (*offset*) i *segment* (każdy segment to 64 kB). Do przechowywania numerów segmentów pamięci komputer IBM PC wykorzystuje tzw. rejestry segmentowe. Konsekwencją takiego rozwiązania są w języku C trzy dodatkowe słowa kluczowe:

- `near` — wskaźniki „bliskie” mają rozmiar 16 bitów i umożliwiają dostęp do danych bieżącego segmentu,
- `far` — wskaźniki „dalekie” obejmują wartości określające przesunięcie i segment, umożliwiając dostęp do dowolnego adresu w pamięci,
- `huge` — wskaźniki „ogromne” to odmiana wskaźników dalekich, zapewniająca możliwość zwiększania i zmniejszania wartości w całym zakresie 1 MB (kompilator generuje odpowiedni kod modyfikujący wartość przesunięcia).

Nie będzie zapewne zaskakujące stwierdzenie, że przetwarzanie programu korzystającego ze wskaźników typu `near` będzie szybsze niż w przypadku programu, w którym zastosowano wskaźniki `far`. Wskaźniki `huge` są oczywiście największym obciążeniem. Kompilatory C wyposażone są w makro zwracające adres odpowiadający podanym wartościom numeru segmentu i przesunięcia:

```
void far *MK_FP(unsigned segment, unsigned offset);
```

Struktury

Język C oferuje technikę grupowania zmiennych pod jedną nazwą, dostarczając w ten sposób wygodnego sposobu przechowywania powiązanych ze sobą informacji i strukturalizowania ich. Składnia definicji struktury jest następująca:

```
typedef struct
{
  typ_zmiennej nazwa_zmiennej;
  typ_zmiennej nazwa_zmiennej;
  .
  .
  .
}
nazwa_struktury;
```

Używanie zmiennych strukturalnych jest niezbędne przy korzystaniu z plików, w których występuje uporządkowanie oparte na rekordach danych. W poniższym przykładzie operować będziemy na prostym pliku z listą adresów. Rozpoczniemy od deklaracji struktury `dane`, złożonej z sześciu pól: `nazwisko`, `adres`, `miasto`, `wojewodztwo`, `poczta` i `nrtelefonu`:

```
typedef struct
{
  char nazwisko[30];
  char adres[30];
  char miasto[30];
  char wojewodztwo[30];
  char kod[6];
  char nrtelefonu[15];
}
dane;
```

Odwołania do pól zmiennej strukturalnej mają postać:

```
zmienna_strukt.nazwa_pola;
```

Nie ma ograniczenia liczby pól struktury, nie jest również wymagane, aby typy pól były takie same lub podobne, na przyklad:

```
typedef struct
{
  char nazwisko[30];
  int wiek;
  char *notatki;
}
dp;
```

Jest to poprawna deklaracja struktury obejmująca: pole tablicy znakowej, pole liczby całkowitej i pole wskaźnika do zmiennej znakowej. Aby przekazać zmienną strukturalną jako parametr, korzystamy z jej adresu — poprzedzamy nazwę zmiennej operatorem `&`. Oto przykładowy program wykorzystujący struktury w celu wykonania prostych operacji na pliku listy adresów:

```
#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <string.h>
#include <fcntl.h>
#include <sys\stat.h>

/* liczba_wierszy to liczba wierszy ekranu */
#define liczba_wierszy 25

typedef struct
{
  char nazwisko[30];
  char adres[30];
  char miasto[30];
  char wojewodztwo[30];
  char kod[6];
  char nrtelefonu[15];
}
dane;

dane rekord;
int handle;


/* Prototypy funkcji */

void ADD_REC(void);
void CLS(void);
void DISPDATA(void);
void FATAL(char *);
void GETDATA(void);
void MENU(void);
void OPENDATA(void);
int SEARCH(void);

void CLS()
{
  int n;

  for(n = 0; n < liczba_wierszy; n++)
    puts("");
}

void FATAL(char *blad)
{
  printf(" \nBlad krytyczny: %s",blad);
  exit(0);
}

void OPENDATA()
{
  /* Sprawdź czy istnieje plik danych. Jeżeli nie, utwórz. */
  /* Jeżeli tak, otwórz do odczytu-zapisu na końcu pliku. */

  handle = open("address.dat",O_RDWR|O_APPEND,S_IWRITE);
```

```
  if (handle == -1)
  {
    handle = open("address.dat",O_RDWR|O_CREAT,S_IWRITE);
    if (handle == -1)
      FATAL("Nie można utworzyć pliku danych");
  }
}

void GETDATA()
{
  /* Pobierz dane adresowe */

  CLS();

  printf("Nazwisko ");
  gets(rekord.nazwisko);
  printf("\nAdres ");
  gets(rekord.adres);
  printf("\nMiasto ");
  gets(rekord.miasto);
  printf("\nWojewództwo ");
  gets(rekord.wojewodztwo);
  printf("\nKod pocztowy ");
  gets(rekord.kod);
  printf("\nNumer telefonu ");
  gets(rekord.nrtelefonu);
}

void DISPDATA()
{
  /* Wyświetl dane adresowe */
  char tekst[5];

  CLS();

  printf("Nazwisko %s",rekord.nazwisko);
  printf("\nAdres %s",rekord.adres);
  printf("\nMiasto %s",rekord.miasto);
  printf("\nWojewództwo %s",rekord.wojewodztwo);
  printf("\nKod pocztowy %s",rekord.kod);
  printf("\nNumer telefonu %s\n\n",rekord.nrtelefonu);

  puts(" Wciśnij ENTER");
  gets(tekst);
}

void ADD_REC()
{
  /* Dołącz do pliku danych nowy rekord */
  int wynik;

  wynik = write(handle,&rekord,sizeof(dane));

  if (wynik == -1)
    FATAL("Zapis do pliku danych niemożliwy");
}

int SEARCH()
{
```

```
  char tekst[100];
  int wynik;

  printf("Wprowadź wzór wyszukiwania ");
  gets(tekst);
  if (*tekst == 0)
    return(-1);

  /* Zlokalizuj początek pliku */
  lseek(handle,0,SEEK_SET);

  do
  {
    /* Załaduj rekord do pamięci */
    wynik = read(handle,&rekord,sizeof(dane));
    if (wynik > 0)
    {
      /* Przeszukaj rekord */
      if (strstr(rekord.nazwisko,tekst) != NULL)
        return(1);
      if (strstr(rekord.adres,tekst) != NULL)
        return(1);
      if (strstr(rekord.miasto,tekst) != NULL)
        return(1);
      if (strstr(rekord.wojewodztwo,tekst) != NULL)
        return(1);
      if (strstr(rekord.kod,tekst) != NULL)
        return(1);
      if (strstr(rekord.nrtelefonu,tekst) != NULL)
        return(1);
    }
  }
  while(wynik > 0);
  return(0);
}

void MENU()
{
  int opcja;
  char tekst[10];

  do
  {
    CLS();
    puts("\n\t\t\tWybierz opcję");
    puts("\n\n\t\t\t1 Dodaj nowy rekord");
    puts("\n\n\t\t\t2 Przeszukiwanie danych");
    puts("\n\n\t\t\t3 Wyjście");
    puts("\n\n\n\n\n");
    gets(tekst);
    opcja = atoi(tekst);

    switch(opcja)
    {
      case 1 : GETDATA();
        /* Przed dołączaniem rekordu przejdź do końca pliku */
        lseek(handle,0,SEEK_END);
```

```
        ADD_REC();
        break;

      case 2 : if (SEARCH())
          DISPDATA();
        else
        {
          puts("NIEZNALEZIONE!");
          puts("Wciśnij ENTER");
          gets(tekst);
        }
        break;

      case 3 : break;
    }
  }
  while(opcja != 3);
}

void main()
{
  CLS();
  OPENDATA();
  MENU();
}
```

Pola bitowe

Język C przewiduje możliwość korzystania w strukturach ze zmiennych o rozmiarze mniejszym niż 8 bitów. Określa się je mianem *pól bitowych*, a ich rozmiar może być dowolny, od 1 bitu wzwyż. Deklaracja pola bitowego wygląda następująco:

```
typ nazwa : liczba_bitów;
```

Przykładem może być deklaracja kilku jednobitowych znaczników stanu:

```
typedef struct
{
  unsigned przeniesienie : 1;
  unsigned zero          : 1;
  unsigned przepelnienie : 1;
  unsigned parzystosc    : 1;
}
df;

df znaczniki;
```

Zmienna `znaczniki` będzie zajmować w pamięci tylko 4 bity, mimo że składa się z 4 pól, z których każde dostępne jest jako osobne pole struktury.

Union

Kolejnym ułatwieniem języka C, pozwalającym zapewnić optymalne wykorzystanie dostępnej pamięci, jest struktura `union`, czyli zbiór zmiennych, współużytkujących jeden adres pamięci. Oznacza to, oczywiście, że w danym momencie dostępna jest tylko jedna ze zmiennych składowych. Deklaracja `union` ma następującą postać:

```
union nazwa
{
  typ nazwa_zmiennej;
  typ nazwa_zmiennej;
  .
  .
  .
  typ nazwa_zmiennej;
};
```

Wyliczenia

Wyliczenie (enumeracja) to przypisanie liście symboli rosnących wartości całkowitych. Wyliczenie deklarujemy:

```
enum nazwa { lista } lista_zmiennych;
```

Przykładem może być definicja listy kolorów:

```
enum KOLORY
{
  CZARNY,
  NIEBIESKI,
  ZIELONY,
  CZERWONY,
  BRAZOWY,
  JASNOSZARY,
  CIEMNOSZARY,
  JASNONIEBIESKI,
  JASNOZIELONY,
  JASNOCZERWONY,
  ZOLTY,
  BIALY
};
```

Operacje na plikach

W operacjach dostępu do plików język C posługuje się buforowanymi strumieniami plikowymi. Niektóre z platform języka, jak UNIX i DOS, oferują również niebuforowane uchwyty plików.

Strumienie buforowane

Dostęp do strumieni buforowanych realizowany jest za pośrednictwem wskaźnika do zmiennej typu `FILE`. Ten szczególny typ danych zdefiniowany został w nagłówku *stdio.h*. Aby więc zadeklarować wskaźnik do pliku, wprowadzamy:

```
#include <stdio.h>

FILE *ptr;
```

Aby otworzyć strumień, używamy funkcji `fopen()`. Pobiera ona dwa parametry: nazwę otwieranego pliku oraz tryb dostępu. Oto lista trybów dostępu.

Tryb	Opis
r	otwórz tylko do odczytu (plik musi istnieć),
w	utwórz do zapisu; zastąp, jeżeli plik o podanej nazwie istnieje,
a	otwórz do dołączania danych (dopisywania na końcu pliku); utwórz nowy plik, jeżeli plik o podanej nazwie nie istnieje,
r+	otwórz istniejący plik do odczytu i zapisu (plik musi istnieć),
w+	utwórz do odczytu i zapisu; zastąp, jeżeli istnieje,
a+	otwórz do czytania i dołączania danych; utwórz nowy plik, jeżeli nie istnieje.

Aby określić tryb tekstowy lub binarny, do opisu trybu można dołączyć `t` lub `b`. W przypadku pominięcia tego znacznika strumień zostanie otwarty w trybie określanym zmienną globalną `_fmode`. Odczyt i zapis danych do strumieni plikowych w trybie tekstowym wiąże się z konwersją — podczas zapisu znaki CR i LF zamieniane są na pary CR LF, a przy odczycie pary CR LF ulegają zamianie na pojedynczy znak LF. Tego rodzaju operacje nie są wykonywane w trybie binarnym.

Jeżeli funkcja `fopen()` nie będzie mogła otworzyć pliku, zwróci w miejsce wskaźnika wartość `NULL` (zdefiniowaną w *stdio.h*). Poniższy program utworzy nowy plik *dane.txt* i udostępni go do odczytu i zapisu:

```
#include <stdio.h>

void main()
{
  FILE *fp;

  fp = fopen("dane.txt","w+");
}
```

Aby zamknąć strumień, używamy funkcji `fclose()`, wymagającej podania wskaźnika do pliku.

```
fclose(fp);
```

Jeżeli podczas zamykania strumienia wystąpi błąd, funkcja `fclose()` zwróci wartość niezerową (znacznik EOF — End Of File). Do przesyłania i odbierania danych ze strumieni służą cztery podstawowe funkcje: `fgetc()`, `fputc()`, `fgets()` i `fputs()`. Funkcja `fgetc()` odczytuje pojedynczy znak z określonego strumienia wejściowego (przekształcany do liczby całkowitej):

```
int fgetc(FILE *fp);
```

Jej odwrotnością jest `fputc()`, zapisująca pojedynczy znak do określonego strumienia wyjściowego:

```
int fputc( int c, FILE *fp);
```

Funkcja `fgets()` odczytuje ze strumienia wejściowego ciąg:

```
char *fgets(char *s, int liczba_bajtów, FILE *fp);
```

Odczyt zostaje przerwany po pobraniu *liczba_bajtów-1* znaków lub znaku nowego wiersza (również wstawianego do tablicy). Do odczytanego ciągu *s* dołączany jest kończący znak NULL (znany także pod postacią '\0'). W przypadku wystąpienia błędów funkcja zwraca NULL.

Funkcja fputs() zapisuje do strumienia ciąg zakończony znakiem NULL (inaczej '\0'):

```
int fputs(const char *s, FILE *fp);
```

Wszystkie opisywane funkcje zwracają w przypadku błędów wartość EOF (zdefiniowaną w *stdio.h*) z wyjątkiem funkcji fgets(), która w przypadku wystąpienia błędu zwraca NULL. Poniższy program tworzy kopię pliku *dane.dat*, o nazwie *dane.old*, ilustrując zarazem użycie wszystkich czterech funkcji:

```
#include <stdio.h>

int main()
{
  FILE *in;
  FILE *out;

  in = fopen("data.dat","r");

  if (in == NULL)
  {
    puts("\nNie można otworzyć do odczytu pliku dane.dat");
    return(0);
  }

  out = fopen("dane.old","w+");

  if (out == NULL)
  {
    puts("\nNie można utworzyć pliku dane.old");
    return(0);
  }

  /* Powtarzaj odczytywanie i zapisywanie pojedynczych bajtów */
  /* aż do natrafienia na EOF  */
  while(!feof(in))
    fputc(fgetc(in),out);

  /* Zamknij strumienie plikowe */
  fclose(in);
  fclose(out);

  return(0);
}
```

W kolejnym przykładowym programie używamy funkcji fputs do kopiowania tekstu ze strumienia stdin (zazwyczaj oznacza to znaki wprowadzane z klawiatury) do nowego pliku *dane.txt*:

```
#include <stdio.h>

int main()
{
```

```
  FILE *fp;
  char tekst[100];

  fp = fopen("dane.txt","w+");

  do
  {
    gets(tekst);
    fputs(tekst,fp);
  }
  while(*tekst);

  fclose(fp);
}
```

Swobodny dostęp do danych strumieni

Dostęp swobodny do danych dostarczanych za pośrednictwem strumieni zapewnia funkcja `fseek()` o prototypie:

```
int fseek(FILE *fp, long liczba_bajtów, int zacznij_od);
```

Funkcja zmienia pozycję wskaźnika pliku skojarzonego ze strumieniem otwartym wcześniej przez `fopen()`. Wskaźnik ustawiany jest na *liczba_bajtów* za (lub przed w przypadku wartości ujemnej) pozycją *zacznij_od*. Tą ostatnią może być początek pliku, bieżące położenie wskaźnika lub koniec pliku. Pozycje te symbolizują stałe `SEEK_SET`, `SEEK_CUR` i `SEEK_END`. Udaną operację `fseek()` sygnalizuje zwrócenie wartości `0`. Uzupełnieniem `fseek()` jest funkcja `ftell()`, zwracająca wartość bieżącej pozycji wskaźnika pliku:

```
long int ftell(FILE *fp);
```

Funkcja zwraca pozycję wskaźnika pliku, określoną jako ilość bajtów od początku pliku, lub `-1` w przypadku błędu.

Uchwyty

Uchwyty plików (*handles*) otwiera funkcja `open()` o prototypie:

```
int open(char *nazwa_pliku, int dostęp[, unsigned tryb]);
```

Udaną operację sygnalizuje zwrócenie numeru uchwytu. W pozostałych przypadkach zwracane jest `-1`. Na wartość *dostęp* składają się połączone bitową operacją OR stałe symboliczne, odpowiadające deklaracjom w pliku *fcntl.h*. Różnią się one w zależności od kompilatora. Do typowych należą:

`O_APPEND`	przed każdym zapisem wskaźnik pliku będzie ustawiany na końcu pliku,
`O_CREAT`	jeżeli plik nie istnieje, zostanie utworzony,
`O_TRUNC`	obcina istniejący plik do długości 0 bajtów,
`O_EXCL`	używane w połączeniu z `O_CREAT`,
`O_BINARY`	otwiera plik w trybie binarnym,
`O_TEXT`	otwiera plik w trybie tekstowym.

Po przypisaniu uchwytu pliku za pomocą polecenia `open()` można korzystać z funkcji `read()` i `write()`. Prototyp `read()` jest następujący:

```
int read(int handle, void *buf, unsigned liczba_bajtów);
```

Funkcja podejmuje próbę odczytu podanej liczby bajtów i zwraca liczbę bajtów faktycznie pobranych przez uchwyt pliku. Odczytane dane umieszczane są w bloku pamięci określonym parametrem *buf*. Funkcja *write()* działa podobnie, nie różni się również jej prototyp i sposób generowania wartości zwracanej. Zapisuje ona podaną ilość bajtów z określonego wskaźnikiem bloku pamięci. Pliki otwierane funkcją `open()` zamykamy funkcją `close()`:

```
int close(int handle);
```

Funkcja `close()` zwraca `0` w przypadku operacji udanej, a `-1` w przypadku wystąpienia błędów.

Dostęp swobodny zapewnia funkcja `lseek()`, bardzo podobna do `fseek()`, ale pobierająca jako parametr numer uchwytu, a nie wskaźnik strumienia `FILE`. W poniższym przykładzie wykorzystujemy uchwyt pliku do zapisu danych z `stdin` (czyli klawiatury) do nowego pliku o nazwie *dane.txt*:

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>

int main()
{
  int handle;
  char tekst[100];

  handle = open("dane.txt", O_RDWR|O_CREAT|O_TRUNC,S_IWRITE);

  do
  {
    gets(tekst);
    write(handle, &tekst, strlen(tekst));
  }
  while(*tekst);

  close(handle);
}
```

Przegląd funkcji plikowych

Norma ANSI definiuje związane z plikami operacje we-wy przy użyciu strumieni, opisując różnorodne funkcje. Prototyp funkcji `fopen()` ma postać:

```
FILE *fopen(const char *nazwa, const char *tryb);
```

Funkcja podejmuje próbę otwarcia strumienia łączącego z plikiem o podanej nazwie w określonym trybie. Udana operacja kończy się zwróceniem wskaźnika typu `FILE`. W przypadku niepowodzenia funkcji zwraca `NULL`. Na wcześniejszych stronach przedstawiony został opis parametru *tryb*.

Funkcja `fclose()` służy do zamykania strumienia otwartego wcześniejszym wywołaniem `fopen()`:

```
int fclose(FILE *fp);
```

Udana operacja `fclose()` kończy się opróżnieniem wszystkich buforów pliku i zwróceniem wartości `0`. W przypadku błędów zwracana jest wartość `EOF`.

Wiele komputerów korzysta z buforowanego dostępu do plików. Oznacza to, że dane, zapisywane do strumienia, wstępnie umieszczane są w pamięci, a faktyczny zapis następuje dopiero po przekroczeniu pewnej granicznej ilości bajtów. Jeżeli w czasie, gdy dane nie zostały jeszcze faktycznie zapisane do strumienia, nastąpi awaria zasilania, dane zostaną utracone. Zabezpiecza przed tym funkcja `fflush()`, wymuszająca zapisanie wszystkich danych oczekujących:

```
int fflush(FILE *fp);
```

Jeżeli wywołanie `fflush()` jest udane, związane ze strumieniem bufory zostają opróżnione i zwracana jest wartość `0`. W przypadku błędów funkcja zwraca wartość `EOF`.

Kolejną funkcją jest `ftell()` zwracająca lokalizację wskaźnika pliku:

```
long int ftell(FILE *fp);
```

Funkcja zwraca przesunięcie wskaźnika pliku w stosunku do początku pliku lub `-1L` w przypadku błędów. Przesunięcie wskaźnika pliku do nowej pozycji umożliwia `fseek()`:

```
int fseek(FILE *fp, long offset, int zacznij_od);
```

Funkcja podejmuje próbę przesunięcia wskaźnika pliku o *offset* bajtów od pozycji *zacznij_od*, określonej jedną ze stałych:

`SEEK_SET`	początek pliku,
`SEEK_CUR`	bieżąca pozycja wskaźnika pliku,
`SEEK_END`	koniec pliku.

Przesunięcie (*offset*) może być wartością dodatnią (przesuwanie wskaźnika w stronę końca pliku) lub ujemną (przesuwanie wskaźnika w stronę początku pliku). Aby szybko przenieść wskaźnik do początku pliku i usunąć wcześniejsze odwołania do błędów, C dostarcza funkcji `rewind()`:

```
void rewind(FILE *fp);
```

Funkcja ta działa podobnie jak `fseek(fp,0L,SEEK_SET)`. Jednak `fseek()` usuwa znacznik `EOF`, a `rewind()` dodatkowo wszystkie sygnały błędów. Informacje o błędach funkcji plikowych można pobrać przy użyciu funkcji `ferror()`:

```
int ferror (FILE *fp);
```

Funkcja zwraca wartość niezerową, jeżeli w określonym strumieniu wystąpił błąd. Po sprawdzeniu wartości `ferror()` należy zadbać o usunięcie sygnałów błędów za pomocą funkcji `clearerr()`:

```
void clearerr(FILE *fp);
```

Sprawdzenie, czy spełniony jest warunek osiągnięcia końca pliku, realizuje predefiniowane makro `feof()`:

```
int feof(FILE *fp);
```

Makro zwraca wartość niezerową, gdy dla danego strumienia stwierdzono osiągnięcie końca pliku. W pozostałych przypadkach zwracaną wartością jest `0`.

Dostępnych jest kilka funkcji realizujących odczyt danych ze strumienia plikowego. Pojedyncze znaki można odczytywać funkcją `fgetc()`:

```
int fgetc(FILE *fp);
```

`fgetc()` zwraca wartość ASCII pobranego znaku lub znak `EOF` w przypadku wystąpienia błędu. Odczyt ciągu danych umożliwia funkcja `fgets()`, odczytująca ciąg zakończony znakiem nowego wiersza:

```
char *fgets(char  *s, int n, FILE *fp);
```

W wyniku udanego wywołania funkcji w zmiennej *s* umieszczany jest ciąg zakończony znakiem nowego wiersza lub zawierający *n-1* znaków. Funkcja zachowuje kończący ciąg znak nowego wiersza, dołączając do ciągu *s* bajt `NULL`. W przypadku nieudanego wywołania zwracany jest wskaźnik pusty. Ciągi zapisujemy do strumienia funkcją `fputs()`:

```
int fputs(const char *s, FILE *fp);
```

Funkcja `fputs()` zapisuje wszystkie znaki ciągu *s*, z wyjątkiem końcowego bajtu `NULL`, do strumienia *fp*. Standardowo funkcja zwraca ostatni zapisany znak, a w przypadku wystąpienia błędów — `EOF`. Dostępna jest również funkcja zapisująca do strumienia pojedynczy znak `fputc()`:

```
int fputc(int c, FILE *fp);
```

Funkcja zwraca zapisany znak lub, w przypadku wystąpienia błędów, znak `EOF`.

Aby odczytać ze strumienia duży blok danych lub rekord, można posłużyć się funkcją `fread()`:

```
size_t fread(void *ptr, size_t rozmiar, size_t n, FILE *fp);
```

Funkcja podejmuje próbę odczytu *n* elementów, z których każdy ma długość *rozmiar*, ze strumienia plikowego *fp* do bloku pamięci określonego wskaźnikiem *ptr*. Aby ustalić, czy operacja przebiegła bez zakłóceń, korzystamy z funkcji `ferror()`.

Siostrzaną funkcją `fread()` jest `fwrite()`:

```
size_t fwrite(const void *ptr, size_t rozmiar, size_t n, FILE *fp);
```

Funkcja zapisuje *n* elementów o długości *rozmiar* z obszaru pamięci określonego wskaźnikiem *ptr* do strumienia *fp*.

Funkcja `fscanf()` umożliwia odczyt danych formatowanych:

```
int fscanf(FILE *fp, const char *format[,adres ...]);
```

Funkcja zwraca liczbę faktycznie odczytanych pól, a `EOF` w przypadku końca pliku. Poniższy przykład ilustruje użyteczność funkcji `fscanf()` podczas odczytywania ze strumienia liczb:

```
#include <stdio.h>

void main()
{
  FILE *fp;
  int a;
  int b;
  int c;
  int d;
  int e;
  char tekst[100];

  fp = fopen("dane.txt","w+");

  if(!fp)
  {
    perror("Nie można utworzyć pliku");
    exit(0);
  }
  fprintf(fp,"1 2 3 4 5 \"Wiersz liczb\"");

  fflush(fp);

  if (ferror(fp))
  {
    fputs("Błąd przy zapisie strumienia", stderr);
    exit(1);
  }

  rewind(fp);
  if (ferror(fp))
  {
    fputs("Błąd przy przewijaniu strumienia", stderr);
    exit(1);
  }

  fscanf(fp,"%d %d %d %d %d %s", &a, &b, &c, &d, &e, tekst);
  if (ferror(fp))
  {
    fputs("Błąd odczytu ze strumienia", stderr);
    exit(1);
  }

  printf("\nFunkcja fscanf() zwróciła %d %d %d %d %d %s",a,b,c,d,e,tekst);
}
```

Jak łatwo zauważyć, zapis formatowanych danych realizuje funkcja `fprintf()`. Gdy pojawia się potrzeba zapisania położenia wskaźnika pliku i późniejszego jego przywrócenia, można skorzystać z funkcji `fgetpos()` i `fsetpos()`. Pierwsza z nich odczytuje bieżącą pozycję wskaźnika pliku:

```
int fgetpos(FILE *fp, fpos_t *pozycja);
```

Funkcja `fsetpos()` ustawia wskaźniki pliku na określonej pozycji:

```
int fsetpos(FILE *fp, const fpos_t *pozycja);
```

Typ `fpos_t` zdefiniowany został w nagłówku *stdio.h*. Funkcje te są wygodniejsze w użyciu niż `ftell()` i `fseek()`.

Z otwartym już strumieniem można skojarzyć nowy plik. Umożliwia to funkcja `freopen()`:

```
FILE *freopen(const char *nazwa, const char *tryb, FILE *fp);
```

Funkcja zamyka strumień istniejący i podejmuje próbę jego ponownego otwarcia przy użyciu podanej nazwy pliku. Znajduje to zastosowanie przy przekierowywaniu strumieni predefiniowanych `stdin`, `stdout` i `stderr` do pliku lub urządzenia. Przykładowo, gdy pojawia się potrzeba przekierowania wszystkich danych wyjściowych kierowanych do `stdout` na drukarkę, można użyć polecenia:

```
freopen("LPT1","w",stdout);
```

Predefiniowane strumienie we-wy

Wstępnie zdefiniowane zostały trzy strumienie we-wy: `stdin`, `stdout` i `stderr`. Domyślnie `stdin` i `stdout` odpowiadają klawiaturze i monitorowi. Na wielu platformach, w tym systemów DOS i UNIX, dostępna jest możliwość ich przekierowania. Strumień `stderr` domyślnie powiązany jest z monitorem (wyświetlaczem). Praktyka jego przekierowywania nie jest raczej stosowana. Jego podstawowym zadaniem jest zapewnienie możliwości wyświetlania komunikatów błędów, nawet w sytuacji gdy powiązanie standardowego wyjścia (`stdout`) zostało zmienione:

```
fputs("Komunikat o błędzie", stderr);
```

Funkcje `printf()` i `puts()` przekazują dane do strumienia `stdout`. Funkcje `scanf()` i `gets()` pobierają dane ze strumienia `stdin`. Przekierowanie tych strumieni zmienia sposób działania funkcji.

Jako przykład plikowych operacji we-wy na platformie PC, korzystających z możliwości przekierowania strumieni, przedstawimy prosty program przesyłający do strumienia `stdout` zawartość określonego pliku, przedstawioną jako wartości szesnastkowe. Polecenie w postaci:

```
dump nazwa_pliku.xxx > dane_wyjściowe.xxx
```

pozwoli zmienić domyślne powiązanie strumienia `stdout` z monitorem.

```
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
#include <string.h>

main(int argc, char *argv[])
{
  unsigned licznik;
  unsigned char v1[20];
  int f1;
  int x;
  int n;
```

```
  if (argc != 2)
  {
    fputs("\nBŁĄD. Poprawna składnia wywołania: dump f1\n",stderr);
    return(1);
  }

  f1 = open(argv[1],O_RDONLY);

  if (f1 == -1)
  {
    fprintf(stderr, "\nBŁĄD. Nie można otworzyć %s\n",argv[1]);
  return(1);
  }

  fprintf(stdout,"\nZAWARTOŚĆ PLIKU %s\n\n",strupr(argv[1]));

  licznik = 0;

  while(1)
  {
    /* Wypełnienie bufora zerami */
    memset(v1,0,20);

    /* Pobranie do bufora danych z pliku */
    x = _read(f1,&v1,16);

    /* x = 0 to EOF, x = -1 oznacza błąd */
    if (x < 1)
      break;

    /* Wyprowadź offset w pliku */
    fprintf(stdout,"%06d(%05x) ",licznik,licznik);

    licznik +=16;

    /* Wyprowadź szesnastkowe wartości bajtów z bufora */
    for(n = 0; n < 16; n++)
      fprintf(stdout,"%02x ",v1[n]);

    /* Wyprowadź wartości ASCII bajtów z bufora */
    for(n = 0; n < 16; n++)
    {
      if ((v1[n] > 31) && (v1[n] < 128))
        fprintf(stdout,"%c",v1[n]);
      else
        fputs(".",stdout);
    }

    /* Zakończ znakiem nowego wiersza */
    fputs("\n",stdout);
  }

  /* zakończenie normalne */
  return(0);
}
```

Ciągi

Język C należy do najlepiej wyposażonych w funkcje obsługi ciągów pośród uniwersalnych języków programowania. Ciąg to jednowymiarowa tablica znaków zakończona bajtem zerowym. Ciągi można inicjować dwoma sposobami. Pierwszym jest nadanie im stałej wartości w kodzie programu:

```
int main()
{
  char *p = "System 5";
  char nazwa[] = "Program testowy";
return(0);
}
```

Drugi to utworzenie ciągu w czasie wykonywania programu za pomocą funkcji `strcpy()`:

```
char *strcpy(char *cel,const char *źródło);
```

Funkcja `strcpy()` kopiuje ciąg źródłowy do lokalizacji docelowej, na przykład:

```
#include <stdio.h>

int main()
{
  char nazwa[50];

  strcpy(nazwa,"Servile Software");

  printf("\nWartość ciągu 'nazwa' to %s",nazwa);
return 0;
}
```

Język C umożliwia bezpośredni dostęp do każdego bajtu ciągu:

```
#include <stdio.h>

int main()
{
  char nazwa[50];

  strcpy(nazwa,"Servile Software");

  printf("\nWartość ciągu 'nazwa' to %s",nazwa);

  /* Zastąpienie pierwszego bajtu literą 's' */
  nazwa[0] = 's';

  printf("\nWartość ciągu 'nazwa' to %s",nazwa);
return 0;
}
```

Niektóre kompilatory C wyposażone zostały w funkcje konwersji ciągów do wielkich i małych liter, nie obejmuje ich jednak norma ANSI. W specyfikacji pojawiają się za to funkcje `toupper()` i `tolower()`, zwracające pojedynczy znak (w postaci wartości `int`) zamieniony na literę wielką lub małą. Łatwo na tej podstawie utworzyć własne funkcje konwersji ciągów:

```
#include <stdio.h>

void strupr(char *zrodlo)
{
  char *p;

  p = zrodlo;
  while(*p)
  {
if((*p)>=97 && (*p)<=122)
    *p = toupper(*p);
    p++;
  }
}

void strlwr(char *zrodlo)
{
  char *p;

  p = zrodlo;
  while(*p)
  {
if((*p)>=65 && (*p)<=90)
    *p = tolower(*p);
    p++;
  }
}

int main()
{
  char nazwa[50];

  strcpy(nazwa,"Servile Software");

  printf("\nWartość ciągu 'nazwa' to %s",nazwa);

  strupr(nazwa);

  printf("\nWartość ciągu 'nazwa' to %s",nazwa);

  strlwr(nazwa);

  printf("\nWartość ciągu 'nazwa' to %s",nazwa);
return 0;
}
```

(To niezupełnie tak. Funkcje toupper i tolower tworzą litery wielkie i małe nie sprawdzając, jaka jest postać źródłowa. Konwersja odbywa się odpowiednio poprzez odjęcie lub dodanie do wartości znaku 32 (bo taka jest różnica pomiędzy odpowiadającymi sobie literami wielkimi i małymi). Jeśli argument funkcji toupper będzie już literą wielką, wynik konwersji okaże się bezsensowny. W tym przypadku ‘S’ zostanie zamienione na ‘3’). W funkcjach strlwr i strupr potrzebne byłoby więc sprawdzanie, czy znak spełnia kryteria, np.:

```
while (*p)
{
if((*p)>=97 && (*p)<=122)
      *p = toupper(*p);
p++;
```

```
}
oraz
while (*p)
{
if((*p)>=65 && (*p)<=90)
      *p = tolower(*p);
p++;
}

Dodatkowe linie programu wstawiłem w kod P.B.).
```

W przeciwieństwie do innych języków programowania C nie narzuca ograniczenia długości ciągu. Jednak w przypadku niektórych procesorów (CPU) pojawia się ograniczenie wielkości bloku pamięci. Oto prosty program odwracający kolejność znaków w ciągu:

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


char *strrev(char *s)
{
  /* Odwraca kolejność znaków w ciągu, pozostawiając jedynie */
  /* końcowy znak NULL */

  char *pocz;
  char *koniec;
  char tmp;

  /* Ustaw wskaźnik 'koniec' na ostatni znak ciągu */
  koniec = s + strlen(s) - 1;

  /* Zabezpiecz wskaźnik do początku ciągu */
  pocz = s;

  /* Zamiana */
  while(koniec >= s)
  {
    tmp = *koniec;
    *koniec = *s;
    *s = tmp;
    koniec--;
    s++;
  }
  return(pocz);
}


void main()
{
  char tekst[100];
  char *p;

  strcpy(tekst,"To jest ciąg");

  p = strrev(tekst);

  printf("\n%s",p);
}
```

strtok()

Funkcja `strtok()` jest istotną funkcją języka C, służącą do wyłączania fragmentów ciągu. Stosuje się ją, gdy poszczególne podciągi rozdzielone są znanymi ogranicznikami, na przykład przecinkami:

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

void main()
{
  char dane[50];
  char *p;

  strcpy(dane,"CZERWONY,POMARAŃCZOWY,ŻÓŁTY,ZIELONY,NIEBIESKI");

  p =  strtok(dane,",");
  while(p)
  {
    puts(p);
    p =  strtok(NULL,",");
  };
}
```

Program można oprzeć też na pętli `for()`:

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

void main()
{
  char dane[50];
  char *p;

  strcpy(dane,"CZERWONY,POMARAŃCZOWY,ŻÓŁTY,ZIELONY,NIEBIESKI");

  for( p = strtok(dane,","); p; p =  strtok(NULL,","))
  {
    puts(p);
  };
}
```

W pierwszym wywołaniu funkcji `strtok()` podajemy nazwę zmiennej ciągu oraz ogranicznik. Funkcja zwraca wówczas wskaźnik do początku pierwszego podciągu i zastępuje pierwszy ogranicznik zerem. Kolejne wywołania `strtok()` wykonywane są w pętli. Pierwszym parametrem jest wówczas `NULL`, a funkcja zwraca kolejne podciągi. Ponieważ dopuszczalne jest podanie listy ograniczników, funkcja `strtok()` może posłużyć do utworzenia prostego programu zliczającego słowa:

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

void main(int argc, char *argv[])
{
  FILE *fp;
  char bufor[256];
```

```
  char *p;
  long licznik;

  if (argc != 2)
  {
    fputs("\nBŁĄD. Poprawna składnia wywołania: wordcnt f1\n",stderr);
    exit(0);
  }

  /* Otwórz plik do odczytu */
  fp = fopen(argv[1],"r");

  /* Sprawdź czy plik został otwarty */
  if (!fp)
  {
    fputs("\nBŁĄD. Nie można otworzyć pliku źródłowego\n",stderr);
    exit(0);
  }

  /* Inicjuj licznik */
  licznik = 0;

  do
  {
    /* Odczytaj z pliku wiersz danych */
    fgets(bufor, 255, fp);

    /* Sprawdź czy nie wystąpił błąd lub znak EOF */
    if (ferror(fp)  || feof(fp))
      continue;

    /* Zlicz słowa w pobranym wierszu */
    /* Słowa wyróżnia się jako elementy rozdzielone znakami */
    /* \t (tab) \n (nowy wiersz) , ; : . ! ? ( ) - spacja */
    p =  strtok(bufor, "\t\n,;:.!?()- ");
    while(p)
    {
      licznik++;
      p = strtok(NULL,"\t\n,;:.!?()- ");
    }
  }
  while(!ferror(fp) && !feof(fp));

  /* Odczyt zakończony. Błąd? */
  if (ferror(fp))
  {
    fputs("\nBłąd przy odczycie pliku źródłowego\n",stderr);
    fclose(fp);
    exit(0);
  }

  /* Odczyt zakończony poprawnie, znakiem EOF */
  /* Wyprowadzamy liczbę słów */
  printf("\nPlik %s zawiera %ld słów(słowa)\n",argv[1],licznik);
  fclose(fp);
}
```

Zamiana liczb na ciągi i ciągów na liczby

Wszystkie kompilatory C zapewniają możliwość konwertowania liczb na ciągi przy użyciu takich funkcji jak `sprintf()`. Funkcja ta ma jednak wiele zastosowań, co powoduje, że jest rozbudowana i mało wydajna. Może ją zastępować funkcja `ITOS()`, korzystająca z dwóch parametrów: liczby całkowitej ze znakiem i wskaźnika do ciągu znakowego. Funkcja kopiuje liczbę do określonego wskaźnikiem miejsca w pamięci. Podobnie jak `sprintf()`, funkcja `ITOS()` nie sprawdza, czy ciąg docelowy ma wystarczającą do przechowania wyniku konwersji długość. Oto przykładowa funkcja, która kopiuje liczbę `signed int` do ciągu znakowego.

```
void ITOS(long x, char *ptr)
{
  /*Zamień dziesiętną liczbę całkowitą ze znakiem na ciąg znaków */
  long pt[9] = { 100000000, 10000000, 1000000, 100000, 10000, 1000, 100, 10, 1 };
  int n;

  /* Sprawdź znak */
  if (x < 0)
  {
    *ptr++ = '-';
    /* Zamień x na wartość bezwzględną */
    x = 0 - x;
  }

  for(n = 0; n < 9; n++)
  {
    if (x > pt[n])
    {
      *ptr++ = '0' + x / pt[n];
      x %= pt[n];
    }
  }
*ptr='\0';
 (zapewnia zakończenie łańcucha znakowego i zapobiega wypisywaniu głupot, gdy
 ➥ zmienna tablicowa ma większy wymiar, niż liczba tego potrzebuje P.B.)
  return;
}

(Powyższy program działa nieprawidłowo, gdy w zamienianej liczbie znajdują się
➥ zera. Poniżej przedstawiam proponowaną przeze mnie poprawną wersję P.B.):

void ITOS(long x, char *ptr)
{
  /*Zamień dziesiętną liczbę całkowitą ze znakiem na ciąg znaków */
  long pt[9] = { 100000000, 10000000, 1000000, 100000, 10000, 1000, 100, 10, 1 };
  int n;
  int licznik=0; //licznik potrzebny do zliczania zer na początku ciągu

  /* Sprawdź znak */
  if (x < 0)
  {
    *ptr++ = '-';
    licznik++;
    /* Zamień x na wartość bezwzględną */
    x = 0 - x;
  }
```

```
    for(n = 0; n < 9; n++)
    {
    licznik++;
        *ptr++ = '0' + x / pt[n];
        x %= pt[n];
    }
    *ptr='\0';
    ptr=ptr-licznik; //powrót wskaźnika na początek ciągu
    licznik=0;
    if(*ptr=='-') //ominięcie minusa na początku (jeśli jest)
           ptr++;
    while(*ptr=='0') {  //pomijanie początkowych zer
           licznik++;
           ptr++;
    }
    while(*ptr!='\0') {
           *(ptr-licznik)=*ptr; //przepisywanie ciągu już bez zer na początku
         ptr++;
    }
    *(ptr-licznik)='\0';

    return;
}
```

Język C oferuje dwie funkcje do zamiany ciągów znakowych na liczby zmiennoprzecinkowe: `atof()` i `strtod()`. Prototyp funkcji `atof()` ma postać:

```
double atof(const char *s);
```

a prototyp funkcji `strtod()`:

```
double strtod(const char *s, char **endptr);
```

Obie funkcje przeglądają ciąg i przeprowadzają konwersję aż do momentu natrafienia na niezrozumiały znak. Różnica między nimi polega na tym, że `strtod()` pobiera dodatkowy parametr, wskaźnik `char` ustawiany na pierwszy znak ciągu, który nie został objęty konwersją. Znacznie zwiększa to wygodę sprawdzania poprawności wykonania operacji.

Aby zamienić ciąg na wartość całkowitą, można użyć funkcji `atoi()`:

```
int atoi(const char *s);
```

Należy pamiętać, że funkcja `atoi()` nie zapewnia żadnej kontroli przepełnienia zmiennej. Nie jest zdefiniowana wartość zwracana w takiej sytuacji. W podobny sposób działa funkcja `atol()`, zwracająca wartość `long`. Odpowiedniki z dodatkowym parametrem noszą nazwy `strol()` i `stroul()`.

Obsługa tekstu

Człowiek zapisuje informacje jako pewien „tekst”, złożony ze słów, liczb i znaków przestankowych. Słowa złożone są z liter wielkich i małych, odpowiednio do wymagań gramatyki. Wszystko to sprawia, że komputerowe przetwarzanie tekstu nie jest

zadaniem prostym. Norma ANSI definiuje wiele funkcji przetwarzania ciągów znakowych, które z natury rozpoznają wielkość liter. Oznacza to, że litera „A" rozpoznawana jest jako różna od „a". Jest to pierwsze zagadnienie, którego rozwiązanie musi znaleźć programista pracujący nad programem przetwarzającym tekst. Na szczęście, zarówno kompilatory Borlanda, jak i Microsoftu wyposażone zostały w funkcje obsługi ciągów, które nie rozpoznają wielkości liter.

Taką odmianą funkcji `strcmp()` jest `stricmp()`, a `strncmp()` — `strnicmp()`. Gdy jednak pojawia się kwestia przenośności kodu, niezbędna jest zgodność z ANSI C, co pociąga za sobą napisanie własnych funkcji.

Poniżej przedstawiamy prostą implementację nierozróżniającej wielkości liter odmiany funkcji `strstr()`. Tworzy ona kopie ciągów, zamienia je na wielkie litery i wykonuje standardową operację `strstr()`. Pozwala to określić poszukiwaną wartość przesunięcia i utworzyć wskaźnik do ciągu źródłowego.

```
char *stristr(char *s1, char *s2)
{
  char c1[1000];
  char c2[1000];
  char *p;

  strcpy(c1,s1);
  strcpy(c2,s2);

  strupr(c1);
  strupr(c2);

  p = strstr(c1,c2);
  if (p)
    return s1 + (p - c1);
  return NULL;
}
```

Kolejna funkcja przegląda ciąg `s1`, wyszukując słowo podane jako `s2`. Aby funkcja zwróciła wartość `TRUE`, znalezione musi zostać odrębne słowo, a nie jedynie sekwencja znaków. Wykorzystujemy przygotowaną wcześniej funkcję `stristr()`.

```
int word_in(char *s1, char *s2)
{
  /*zwraca wartość niezerową, jeżeli s2 jest słowem zawartym w s1*/
  char *p;
  char *q;
  int ok;

  ok = 0;
  q = s1;

  do
  {
    /* Lokalizuj wystąpienie sekwencji znaków s2 w s1 */
    p = stristr(q,s2);
    if (p)
    {
```

```
        /* Znaleziony */
        ok = 1;

        if (p > s1)
        {
          /* Sprawdź znak przed znalezionym ciągiem*/
          if (*(p-1) >= 'A' && *(p-1) <= 'z')
            ok = 0;
        }

        /* Niech p wskazuje koniec ciągu */
        p += strlen(s2);

        if (*p)
        {
          /* Sprawdź znak za znalezionym ciągiem */
        if (*p >= 'A' && *p <= 'z')
          ok = 0;
        }
      }
      q = p;
    }
    while(p && !ok);
    return ok;
  }
```

Szerokie zastosowanie znajdzie kilka dalszych prostych funkcji znakowych. `truncstr()` obcina ciąg znakowy:

```
void truncstr(char *p, int liczba)
{
  /* Obcina 'liczba' znaków z ciągu 'p' */
  if (liczba < strlen(p))
    p[strlen(p) - liczba] = 0;
}
```

`trim()` usuwa końcowe znaki spacji (odstępu międzywyrazowego) w ciągu:

```
void trim(char *tekst)
{
  /* usuwa spacje końcowe */
  char *p;

  p = &tekst[strlen(tekst) - 1];
  while(*p == 32 && p >= tekst)
    *p-- = 0;
}
```

`strlench()` zmienia długość ciągu:

```
void strlench(char *p,int num)
{
  /* Zmienia długość ciągu, dołączając lub usuwając znaki */

  if (num > 0)
    memmove(p + num,p,strlen(p) + 1);
  else
  {
```

```
    num = 0 - num;
    memmove(p,p + num,strlen(p) + 1);
  }
}
```

`strins()` umieszcza jeden ciąg w innym:

```
void strins(char *p, char *q)
{
  /* Wstaw ciąg q do ciągu p */
  strlench(p,strlen(q));
  strncpy(p,q,strlen(q));
}
```

`strchg()` zastępuje wszystkie wystąpienia pewnego podciągu innym podciągiem:

```
void strchg(char *dane, char *s1, char *s2)
{
  /* Zastępuje wszystkie wystąpienia s1 ciągiem s2 */
  char *p;
  char zmienione;

  do
  {
    zmienione = 0;
    p = strstr(dane, s1);
    if (p)
    {
      /* Usuń ciąg znaleziony */
      strlench(p, 0 - strlen(s1));

      /* Wstaw ciąg */
      strins(p,s2);
      zmienione = 1;
    }
  }
  while(zmienione);
}
```

Data i godzina

Język C wyposażony jest w funkcję `time()`, która odczytuje zegar systemowy komputera i podaje informację o dacie i godzinie w postaci liczby sekund, która upłynęła od północy 1 stycznia 1970 roku. Wartość ta może zostać zamieniona na czytelny dla człowieka ciąg znaków za pomocą funkcji `ctime()`:

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

int main()
{
  /* Struktura do przechowywania daty i godziny, z time.h */
  time_t t;
  /* Pobierz datę i godzinę systemu */
  t = time(NULL);
  printf("Bieżąca data i godzina: %s\n",ctime(&t));
}
```

Na ciąg zwracany przez `ctime()` składa się siedem pól:

- dzień tygodnia,
- miesiąc roku,
- dzień miesiąca,
- godzina,
- minuty,
- sekundy,
- rok.

Uzupełnieniem jest znak nowego wiersza i końcowe 0. Ponieważ pola mają stałą szerokość, ciąg zwracany przez `ctime()` idealnie nadaje się do operacji wymagających wyodrębnienia elementów daty lub godziny. W poniższym programie definiujemy strukturę `godzina` oraz funkcję `pobierz_godzine()`, której zadaniem jest wypełnienie struktury treścią pól ciągu `ctime()`:

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

struct godzina
{
  int g_min;   /* Minuty */
  int g_godz;  /* Godziny */
  int g_sek;   /* Sekundy */
};

void pobierz_godzine(struct godzina *teraz)
{
  time_t t;
  char temp[26];
  char *ts;

  /* Pobierz datę i godzinę systemu */
  t = time(NULL);

  /* Przedstaw datę i godzinę w postaci ciągu */
  strcpy(temp,ctime(&t));

  /* Obetnij ostatnie pole */
  temp[19] = 0;

  ts = &temp[11];

  /* Przeszukaj ciąg i skopiuj elementy do struktury */
  sscanf(ts, "%2d:%2d:%2d",&teraz->g_godz,&teraz->g_min,&teraz->g_sek);
}

int main()
{
```

```
  struct godzina teraz;

  pobierz_godzine(&teraz);

  printf("\nJest godzina %02d:%02d:%02d",&teraz.g_godz,&teraz.g_min,&teraz.g_sek);
}
```

Norma ANSI przewidziała również funkcję konwertującą wartość zwracaną przez funkcję `time()` do postaci struktury. Przedstawiony poniżej przykład zawiera deklarację struktury `tm` z nagłówka `time.h`:

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

int main()
{
  time_t t;
  struct tm *tb;

  /* Pobierz czas do t */
  t = time(NULL);

  /* Zamień wartość t na strukturę tb */
  tb = localtime(&t);

  printf("\nJest godzina %02d:%02d:%02d",tb->tm_hour,tb->tm_min,tb->tm_sec);
return (0);
}
```

Struktura `tm` (zawarta w pliku *time.h*) ma następującą postać:

```
struct tm
{
  int tm_sec;
  int tm_min;
  int tm_hour;
  int tm_mday;
  int tm_mon;
  int tm_year;
  int tm_wday;
  int tm_yday;
  int tm_isdst;
};
(Ta struktura nie może być częścią programu, jak to zasugerowano, bo jest już
➥ zdefiniowana w pliku nagłówkowym. W takiej sytuacji kompilator wyświetla błąd.
➥ Można ją zostawić w tym miejscu z komentarzem, który podałem na górze, względnie
➥ przenieść na stronę 51 P.B.)
```

Liczniki czasu

Programy często korzystają z możliwości pobrania daty i czasu z nieulotnej pamięci RAM komputera. Norma ANSI przewiduje kilka różnych funkcji, które mogą zostać do tego celu wykorzystane. Pierwszą jest funkcja `time()`, zwracająca liczbę sekund od 1 stycznia 1970 roku:

```
time_t time(time_t *timer);
```

Funkcja wypełnia przekazaną jej jako parametr zmienną typu `time_t` (jeśli nie jest to `NULL`), zwracając tę samą wartość również jako wartość wyjściową. Można więc wywoływać funkcję `time()` z parametrem `NULL` i korzystać z wartości zwracanej:

```
#include <time.h>

void main()
{
  time_t teraz;

  teraz = time(NULL);
}
```

Funkcja `asctime()` zamienia strukturę `tm` na 26-znakowy ciąg (przedstawiony przy opisie funkcji `ctime()`):

```
char *asctime(const struct tm *struktura);
```

Funkcja `ctime()` zamienia wartość czasu (zwracaną przez `time()`) na 26-znakowy ciąg:

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


void main()
{
  time_t teraz;
  char data[30];

  teraz = time(NULL);
  strcpy(data,ctime(&teraz));
}
```

Kolejna funkcja, `difftime()`, zwraca, liczoną w sekundach, różnicę między dwoma wartościami typu `time_t`. Służy więc do wyznaczania ilości czasu, jaki upłynął między dwoma zdarzeniami, czasu wykonywania funkcji lub generowania przerw w pracy programu, na przykład:

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

void DELAY(int okres)
{
  time_t pocz;

  pocz = time(NULL);
  while(time(NULL) < pocz + okres)
    ;
}

void main()
{
  printf("\nRozpoczynam oczekiwanie... (5 sekund)");

  DELAY(5);

  puts("\nOczekiwanie zakończone.");
}
```

Funkcja `gmtime()` zamienia lokalną wartość czasu `time_t` na wartość GMT o postaci struktury `tm`. Działanie tej funkcji zależy od ustawienia globalnej zmiennej strefy czasowej. Struktura `tm` została wstępnie zdefiniowana w nagłówku `time.h`. Przedstawiliśmy ją kilka stron wcześniej.

```
struct tm
{
  int tm_sec;
  int tm_min;
  int tm_hour;
  int tm_mday;
  int tm_mon;
  int tm_year;
  int tm_wday;
  int tm_yday;
  int tm_isdst;
};
```

Element struktury `tm_mday` przechowuje dzień miesiąca (od 1 do 31), a `tm_wday` — dzień tygodnia (gdzie niedzieli odpowiada 0). Czas jest mierzony od 1900 roku. Wartość `tm_isdst` to znacznik, który informuje o tym, czy stosowany jest czas letni. Stosowane nazwy struktury i jej elementów mogą różnić się w zależności od kompilatora, jednak sama struktura zasadniczo pozostaje niezmieniona.

Funkcja `mktime()` zamienia strukturę `tm` na wartość `time_t`, uzupełniając wartości pól `tm_wday` i `tm_yday`:

```
time_t mktime(struct tm *t);
```

W kolejnym przykładzie umożliwiamy wprowadzanie daty i używamy funkcji `mktime()` do ustalenia dnia tygodnia. Należy pamiętać, że funkcje związane z czasem rozpoznają wyłącznie daty późniejsze niż 1 stycznia 1970:

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

void main()
{
  struct tm tstruct;
  int okay;
  char data[100];
  char *p;
  char *wday[] =
{"niedziela","poniedziałek","wtorek","środa","czwartek","piątek","sobota", "przed
➥ rokiem 1970 - nieznany"};
  do
  {
    okay = 0;
    printf("\nWprowadź datę w formacie dd/mm/rr ");
    p = fgets(data, 9,stdin);
    p = strtok(data,"/");

    if (p!= NULL)
      tstruct.tm_mday = atoi(p);
    else
      continue;
```

```
    p = strtok(NULL,"/");
    if (p != NULL)
      tstruct.tm_mon = atoi(p);
    else
      continue;

    p = strtok(NULL, "/");

    if (p != NULL)
      tstruct.tm_year = atoi(p);
    else
      continue;
    okay = 1;
  }
  while(!okay);

  tstruct.tm_hour = 0;
  tstruct.tm_min = 0;
  tstruct.tm_sec = 1;
  tstruct.tm_isdst = -1;

  /* Teraz ustalimy dzień tygodnia */
  if (mktime(&tstruct) == -1)
    tstruct.tm_wday = 7;

  printf ("Ten dzień to %s\n", wday[tstruct.tm_wday]);
}
```

Funkcja `mktime()` zapewnia również wprowadzenie odpowiednich poprawek dla wartości przekraczających swój dopuszczalny zakres. Można to wykorzystać do ustalenia dokładnej daty, odległej o `n` dni:

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

void main()
{
  struct tm *tstruct;
  time_t dzisiaj;

  dzisiaj = time(NULL);
  tstruct = localtime(&dzisiaj);

  tstruct->tm_mday += 10;
  mktime(tstruct);
  if(tstruct->tm_year>99)
         tstruct->tm_year%=100;

(te dwie linie zapobiegają wyświetlaniu błędnego roku w przypadku, gdy czas, jaki
➥ upłynął od roku 1900 jest dłuższy od 99 lat P.B.)

  printf("Za dziesięć dni będzie %02d/%02d/%02d\n",tstruct->tm_mday,tstruct->tm_mon +
  ➥ 1,tstruct->tm_year);
}
```