Język ANSI C

część 4

biblioteka standardowa wejścia - wyjścia
znakowe wejście - wyjście

Jarosław Gramacki
Instytut Informatyki i Elektroniki

Typy dyskowego wejścia - wyjścia

pierwszy podział:

gdyż korzysta ze standardowej biblioteki funkcji we-wy języka C

– **standardowe** we-wy (strumieniowe we-wy):

posiada szeroki zakres poleceń
wydaje się, że jest dosyć łatwe w użyciu
ukrywa przed programistą szczegóły operacji na plikach
(np. buforowanie, automatyczna konwersja danych)

gdyż korzysta z możliwości wywoływania systemowych funkcji we-wyj z poziomu języka C

– **systemowe** we-wy (niskopoziomowe, deskryptorowe we-wy):

węższy zakres poleceń
bardziej zbliżone do systemu operacyjnego na którym pracujemy
programista obsługuje szczegóły operacji plikowych
(np. ręczna obsługa bufora, ręczna konwersja danych gdy jest wymagana)
efektywniejsza, szybsza obsługa plików gdyż pracujemy "bliżej" systemu operacyjnego

Typy dyskowego wejścia - wyjścia

drugi podział:

- **znakowe** we-wy
  dane odczytywane / zapisywane są znak po znaku
- **łańcuchowe** we-wy
  dane odczytywane / zapisywane są całymi wierszami
- **sformatowane** we-wy
  dane odczytywane / zapisywane są analogicznie do `printf()` i `scanf()`
- **rekordowe** (blokowe) we-wy
  dane odczytywane / zapisywane są całymi blokami (rekordami). Z reguły w formacie binarnym

Typy dyskowego wejścia - wyjścia

trzeci podział: format tekstowy / binarny

- często mylone są dwa pojęcia:

  tryb tekstowy / binarny pracy z plikiem - dotyczy sposobu tłumaczenia znaków końca linii i znaku końca pliku

  format tekstowy / binarny zapisu danych na pliku - dotyczy sposobu reprezentacji zapisywanych w pliku danych
- różnica pomiędzy trybem tekstowym, a binarnym polega na innym traktowaniu znaków 1. nowej linii, 2. końca pliku. W trybie tekstowym przejście do nowej linii jest zapisywane w postaci dwóch znaków CR i LF. Przy czytaniu - podobnie, napotkanie CR i LF jest tłumaczone na znak nowej linii. W trybie binarnym taka interpretacja nie zachodzi - kombinacja CR i LF jest traktowana jako dwa znaki.
- do tematu powrócimy po zapoznaniu się z podstawowymi funkcjami we-wy. Wtedy podane zostaną odpowiednie przykłady ilustrujące "problem"

Biblioteka standardowa wejścia - wyjścia

– najbardziej podstawowe funkcje z podziałem na rodzaj wejścia-wyjścia

znakowe we-wy
getc()
getchar()
fgetc()
putc()
putchar()
fputc()
ungetc()

rekordowe we-wy
fread()
fwrite()
fseek()
ftell()
rewind()
fgetpos()
fsetpos()

systemowe we-wy (non-ANSI, Low Level I/O)
open()
close()
read()
write()

łańcuchowe we-wy
gets()
puts()
fgets()
fputs()

formatowane we-wy
scanf()
fscanf()
sscanf()
printf()
fprintf()
sprintf()

operacje na plikach
fopen()
fclose()
fflush()
remove()
rename()
setvbuf()
setbuf()

obsługa błędów
clearerr()
feof()
ferror()

Znakowe we-wy

– wybrane deklaracje z `stdio.h`

```
/* Returned by various functions on end of file condition or error. *
/#define EOF      (-1)

/*
 * The buffer size as used by setbuf such that it is equivalent to
 * (void) setvbuf(fileSetBuffer, caBuffer, _IOFBF, BUFSIZ).
 */
#define  BUFSIZ   512

/* Constants indicating the position relative to which fseek
 * sets the file position. Enclosed in ifdefs because io.h could also
 * define them. (Though not anymore since io.h includes this file now.)
 */

#ifndef  SEEK_SET
#define  SEEK_SET (0)
#endif

#ifndef  SEEK_CUR
#define  SEEK_CUR (1)
#endif

#ifndef  SEEK_END
#define  SEEK_END (2)
#endif
```

Znakowe we-wy

– wybrane deklaracje z `stdio.h`

```
// File Operations

FILE*    fopen (const char* szFileName, const char* szMode);
int      fclose (FILE* fileClose);

int      fflush (FILE* fileFlush);
int      remove (const char* szFileName);
int      rename (const char* szOldFileName, const char* szNewFileName);

int      setvbuf (FILE* fileSetBuffer, char* caBuffer, int nMode,
                  size_t sizeBuffer);
void     setbuf (FILE* fileSetBuffer, char* caBuffer);


// Character / String Input and Output Functions

int      fgetc (FILE* fileRead);
char*    fgets (char* caBuffer, int nBufferSize, FILE* fileRead);
int      fputc (int c, FILE* fileWrite);
int      fputs (const char* szOutput, FILE* fileWrite);
int      getc (FILE* fileRead);
int      getchar ();
char*    gets (char* caBuffer);    // Unsafe: how does gets know how
                                   // long the buffer is?
int      putc (int c, FILE* fileWrite);
int      putchar (int c);
int      puts (const char* szOutput);
int      ungetc (int c, FILE* fileWasRead);
```


Znakowe we-wy

– wybrane deklaracje z `stdio.h`

bardzo ciekawy mechanizm funkcji o nieokreslonej liczbie parametrów

```
// Formatted Output

int      printf (const char* szFormat, ...);
int      fprintf (FILE* filePrintTo, const char* szFormat, ...);
int      sprintf (char* caBuffer, const char* szFormat, ...);

// Formatted Input

int      scanf (const char* szFormat, ...);
int      fscanf (FILE* fileReadFrom, const char* szFormat, ...);
int      sscanf (const char* szReadFrom, const char* szFormat, ...);


// Direct Input and Output Functions

size_t   fread    (void* pBuffer, size_t sizeObject, size_t sizeObjCount,
                   FILE* fileRead);
size_t   fwrite   (const void* pObjArray, size_t sizeObject,
                   size_t sizeObjCount, FILE* fileWrite);
```

Znakowe we-wy

– wybrane deklaracje z `stdio.h`

```
// File Positioning Functions

int      fseek    (FILE* fileSetPosition, long lnOffset, int nOrigin);
long     ftell    (FILE* fileGetPosition);
void     rewind   (FILE* fileRewind);

typedef long     fpos_t;

int      fgetpos  (FILE* fileGetPosition, fpos_t* pfpos);
int      fsetpos (FILE* fileSetPosition, fpos_t* pfpos);


// Error Functions

void     clearerr (FILE* fileClearErrors);
int      feof (FILE* fileIsAtEnd);
int      ferror (FILE* fileIsError);
void     perror (const char* szErrorMessage);
```


Znakowe we-wy

– przykład: pobieraj znaki z klawiatury i zliczaj je

```
#include <stdio.h>

#define RETURN '\n'                 //  \n == return in UNIX
                                    //  \r == return in DOS
int main(void)
{
  int count = 0;
  puts("Please enter some text.");
                                    // Count the letters in 'stdin' buffer.
  while ( getchar() != RETURN )
   count++;

  printf("You entered %d characters\n", count);
  return 0;
}
```

– `getchar()` odpowiada ogólniejszej funkcji `getc(stdin)`

– `stdin` - standardowy wskaźnik (strumień) plikowy skojarzony zawsze z klawiaturą. Udostępniany zawsze automatycznie. Pozostałe standardowe strumienie: `stdout, stderr`

– wprowadzane znaki są buforowane (w buforze klawiatury !) do czasu naciśnięcia ENTER. `getchar()` "obrabia" je dopiero gdy naciśniemy ENTER i wtedy następuje przesłanie zawartości bufora klawiatury do programu w języku C

– zwróć uwagę na `\n` oraz `\r`

Znakowe we-wy

- `getchar()` zdefiniowana jest w `stdio.h` jako makro. Jest to pożyteczne. Unika się w ten sposób narzutu spowodowanego realizacją wywołania funkcji dla każdego przetwarzanego znaku
- makro jest tłumaczone tylko raz (w momencie kompilacji)
- podobne funkcje `getche(), getch()` (nie mają one "problemów" z buforem) NIE są funkcjami standardowymi. W systemie DOS były zdefiniowane w `conio.h`. W systemie Linux są dostępne w bibliotece `ncurses` lub `termios.h`

```
getch
This is a nonstandard function that gets a character
from keyboard, does not echo to screen.

getche
This is a nonstandard function that gets a character
from the keyboard, echoes to screen.
```


Znakowe we-wy, buforowanie

- `getchar()` jest buforowana - to uwaga z poprzedniego slajdu
- jak więc sobie radzić z tym "problemem", gdy buforowanie nie jest nam na rękę ?
- 1. korzystać z dostępnych funkcji niestandardowych:
  `<conio.h>` biblioteka dostarczane z wieloma kompilatorami pod DOS, Windows
  `<ncurses.h>` - biblioteka dostępna w Linux
- 2. spróbować wyłączyć buforowanie: funkcja `setbuff`
- 3. napisać własne rozwiązanie korzystając np. z systemowego we-wy (systemowe we-wy jest tematem oddzielnego wykładu)

```
// niebuforowany odpowiednik getchar()
// czyta w wejścia znak po znaku

#include <io.h> // patrz wykład z systemowego we-wy

int my_getchar (void)
{
   char c;       // c musi być tu typu char !
                 // gdyż read() akceptuje tylko wskaźniki do znaków

   return (read(0, &c, 1) == 1) ? (unsigned char) c : EOF;
}
```

Znakowe we-wy, buforowanie

```
int main(void)
{
char z;
printf("Do dziela...\n");

while ( (z = getch()) != 'q')
//while ( (z = getche()) != 'q')
//while ( (z = my_getchar()) != 'q')
//while ( (z = getchar()) != 'q')
printf("%c \n", z);

return 0;
}
```


Znakowe we-wy

– znakowe wejście-wyjście; przykład: wyświetl zawartość pliku na ekranie dużymi literami

```
#include <stdio.h>
#include <ctype.h> // for toupper()

int main(void)
{
  int c;                    // Character read from the file.
  FILE *pFile;              // Pointer to the file. FILE is a
                            // structure  defined in <stdio.h>

  pFile = fopen ( "/etc/hosts","r" );

                            // Read one character at a time, checking
                            // for the End of File
                            // EOF is defined in <stdio.h>  as -1

  while ((c = fgetc ( pFile )) != EOF )     // all brackets NECESSARY !!!
    printf( "%c", toupper(c) );
    // putchar(toupper(c) );                // or like that
    // putc (toupper(c), stdout );          // or like that

  fclose( pFile );                          // Close the file.
  return 0;
}
```

not char !

int c;

Znakowe we-wy

- `getc() i fgetc()` są sobie równoważne
- funkcje zwracają odczytany ze strumienia następny znak i zwracają jego wartość potraktowaną jako `unsigned char` i przekształconą do `int`. Po napotkaniu końca pliku albo błędu zwracają EOF
- znak EOF to sygnał generowany przez system operacyjny informujący o końcu pliku. Znak EOF NIE występuje w pliku (nie ma przecież znaków o ujemnych kodach !)
- tylko zmienna typu `int` jest w stanie pomieścić wszystkie możliwe znaki (kody 0-255) oraz znacznik końca pliku (-1). Nie sprawdzi się tu zmienna ani `char` (-127 - 128) ani `unsigned char` (0 - 255)

Znakowe we-wy

- inne tryby otwarcia pliku

```
"r" - tylko do odczytu
"w" - tylko do zapisu (jeśli nie istnieje utwórz)
"a" - do odczytu, dane będą dodawane na końcu pliku
"r+"- do odczytu i zapisu
"w+"- do odczytu i zapisu (jeśli nie istnieje utwórz)
"a+" - do odczytu i zapisu, dane będą dodawane na końcu pliku
```

- Uwaga. W systemach, w których rozróżnia się pliki tekstowe i binarne (np. DOS / Windows), dołączyć należy literę "b" (np. "rb" - otwarcie pliku do odczytu w trybie binarnym)

Znakowe we-wy

– przykład: zlicz ilość znaków (bajtów) w pliku
Plik podaj w trakcie wykonania programu

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int count = 0;
  FILE *pFile;
  int c;

  pFile = fopen( argv[1], "r" );

  c = fgetc( pFile );
  while ( c != EOF )
  {
    count++;
    c = fgetc( pFile );
  }

  printf( "%d\n", count );
  fclose( pFile);
  return 0; }
```

`argc`: ilość argumentów wywołania programu (włącznie ze ścieżką do programu)
`*argv[]`: tablica wskaźników na te argumenty (wskaźniki na napisy)

– przykładowe wywołanie: `./countchars /home/jarek/C/plik.c`

– a co gdy: `./countchars /home/jarek/C/plik` (brak takiego pliku) ?

– a co gdy: `./countchars` (brak argumentu) ?

Znakowe we-wy

– wersja z obsługą błędów

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

int main(int argc,char *argv[])
{
  int count = 0;
  FILE * pFile;
  int c;

  if (argc < 2) {
    printf("Usage: countchars <file name>\n");
    exit(1);
  }

 pFile = fopen(argv[1], "r");

  if ( pFile == NULL ) {
   printf("File %s doesn't exist\n", argv[1]);
   exit(1);
  }

  c = fgetc( pFile );
  while ( c != EOF ) {
    count++;
    c = fgetc (pFile);
  }
  printf("%d\n", count); }
```

funkcja `fopen` tworzy strukturę `FILE`, wypełnia informacjami jej pola i zwraca wskaźnik na tą strukturę. Jeżeli pliku nie uda się otworzyć, to zwraca wskaźnik `NULL`

– wyniki nie zgadzają się z poleceniem `dir` (DOS, Windows). Na przykład program wyświetli liczbę 498 a z polecenia dir odczytujemy 524 ?

Znakowe we-wy

– problem trybu otwarcia pliku (końca **wiersza**)

```
infile = fopen(argv[1], "rb");   // in DOS, Windows
                                 // with b - binary mode
                                 // without b - text mode


plik w DOS/Windows:

adadadadaa\r\n
adadada adadad adadad\r\n                // '\r' '\n'  CR/LF


plik w Linux/Unix:

adadadadaa\n
adadada adadad adadad\n
```

analogiczny problem występuje przy transmisji plików protokołem FTP. Polecenia protokołu `ascii` i `binary` włączają odpowiednio tryb transmisji tekstowy i binarny

– problem o podłożu historycznym

– "wewnątrz" programu w C (działającego w WIN lub Unix) pliki przechowywane są zawsze w konwencji Unix

– w trybie tekstowym (otwarcie pliku w trybie tekstowym) znak nowej linii `\n` jest tłumaczony na kombinację `\r\n` przed zapisem na dysk. Podobnie kombinacja `\r\n` na dysku jest tłumaczona na `\n` przy odczycie pliku przez program w C

– w trybie binarnym takie tłumaczenie NIE następuje

Znakowe we-wy

– problem znaku końca **pliku.**

– w trybie tekstowym i binarnym inaczej obsługiwany jest znak końca pliku

– system operacyjny śledzi długość pliku (bo ta informacja jest znana systemowi operacyjnemu) i sygnalizuje wystąpienie końca pliku po dojściu do jego końca.

– w trybie tekstowym odczytywany jest znak Ctrl-Z (znak o kodzie dziesiętnym 26 lub 1A szesnastkowo). W trybie binarnym śledzona jest całkowita długość pliku, a koniec pliku sygnalizowany wartością całkowitą -1 (FFFFH). Wartość ta jest zdefiniowana w postaci symbolicznej jako EOF (od ang. end of file). Jeżeli więc w pliku występują dane w formacie binarnym lub tekst w innym kodzie niż ASCII, to w trybie tekstowym, przypadkowe wystąpienie bajtu o wartości 1A, zostanie potraktowane jako koniec pliku !

Znakowe we-wy

```
jarek@mykonos:~/C/testy$ ./a.out bindump_DOS.c

 2f  2f  20  65  61  63  68  20  6c  69     // each li
 6e  65  20  69  73  20  74  65  6e  20     ne is ten
 41  53  43  49  49  20  63  6f  64  65     ASCII code
 73  20  66  6f  6c  6c  6f  77  65  64     s followed
 20  62  79  20  74  65  6e  20  63  68      by ten ch
 61  72  61  63  74  65  72  73   d   a     aracters..
 23  69  6e  63  6c  75  64  65  20  3c     #include <
 73  74  64  69  6f  2e  68  3e  20  20     stdio.h>
 20  20  20  20  20  20  20  20  20  20
 20  20  20  20  20  20  20  20   d   a                ..
 23  69  6e  63  6c  75  64  65  20  3c     #include <
 73  74  64  6c  69  62  2e  68  3e  20     stdlib.h>
 20  20  20  20  20  20  20  20  20  20
 20  20  20  20  20  20  20  20   9   9                ..
  9  2f  2f  20  66  6f  72  20  65  78     .// for ex
 69  74  28  29   d   a  23  64  65  66     it()..#def
 69  6e  65  20  4c  45  4e  47  54  48     ine LENGTH
 20  31  30  20  20  20  20  20  20  20      10
...
 6e  20  45  4f  46   d   a  20  20  20     n EOF..
 66  63  6c  6f  73  65  28  66  70  74     fclose(fpt
 72  29  3b  20  20  20  20  20  20  20     r);
 20  20  20  20  20  20  20  20  20  20
 20  20   9   9   9  2f  2f  20  63  6c       ...// cl
 6f  73  65  20  66  69  6c  65   d   a     ose file..
 20  20  20  72  65  74  75  72  6e  28        return(
 30  29  3b   d   a  20  20  20  7d   d     0);..   }.
  a ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
```

d a gdyż program bindump.c pisany był w Windows

Na końcu program zaczyna czytać znaki EOF (-1, FFFFFFFF) aż zakończy się pętla for(j=0; j<LENGTH; j++). Popraw ten błąd samodzielnie.

Znakowe we-wy

```
// each line is ten ASCII codes followed by ten characters
#include <stdio.h>
#include <stdlib.h>                                     // for exit()
#define LENGTH 10                                       // length of display line
#define TRUE    1
#define FALSE   0

int main( int argc, char *argv[] )
   {
   FILE *pFile;
   int ch;
   int j, not_eof;
   unsigned char string[LENGTH+1];                      // buffer for chars

   if(argc != 2)                                        // check arguments
      { printf("Format: ./bindump filename"); exit(1); }
   if( (pFile = fopen(argv[1], "rb"))==NULL )           // binary read
      { printf("Can't open file %s", argv[1]); exit(1); }
   not_eof = TRUE;                                      // not EOF flag
   do
      {
      for(j=0; j<LENGTH; j++)                           // chars in one line
         {
           if( (ch = getc(pFile)) == EOF )              // read character
              not_eof = FALSE;                          // clear flag on EOF
           printf("%3x ", ch);                          // print ASCII code
         if (ch > 31)
              string[j] = ch;                           // save printing char
           else                                         // use period for
              string[j] = '.';                          // non-printing char
         }
      string[j] = '\0';                                 // end string
      printf("   %s\n", string);                        // print string
      }
   while(not_eof == TRUE);                              // quit on EOF
   fclose(pFile);                                       // close file
   return(0);          }
```

- znakowe we-wy

```
jarek@mykonos:~/C/testy$ ls -la
total 352
...
-rw-r--r--   1 jarek staff   1494 Nov  9 01:08 bindump_DOS.c
-rw-r--r--   1 jarek staff   1454 Nov  9 01:00 bindump_Linux.c
...
```


Znakowe we-wy

– przykład: kopia pliku (polecenie `cp`)

– w jakim trybie otworzyć plik do odczytu w systemie WIN aby "było dobrze" ?
(uruchomienie programu w systemie 1. Linux, 2. Windows)

```
#include <stdio.h>

int main(void)
{
  int c;

  FILE *IFile;
  FILE *OFile;

  IFile = fopen("/etc/hosts", "r");// Open the file - no error checking done
  OFile = fopen("/tmp/hosts", "w");

  while ((c = fgetc(IFile)) != EOF)
    fputc(c, OFile);              // send a character to the file


  fclose(IFile);
  fclose(OFile);

return 1; }
```

– `fgetc` i `getc` oraz `fputc` i `putc` są sobie równoważne

– `putc(), getc()` są w większości systemów implementowane jako makro

Znakowe we-wy

– dodanie obsługi błędów + pliki jako argumenty programu
+ wyodrębniona funkcja kopiująca

```
int copyfile (const char *IFile, const char *OFile) {
...
if (( IFile = fopen(IFile, "r")) == NULL)
    return(-1);

if (( OFile = fopen((OFile, "w")) == NULL)
    {
      fclose(IFile);
      return(-2);
    }
...
return 0;
}

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

     if(argc<3)
     {
         printf( "Use: %s source_file dest_file", argv[0] );
         exit(1);
     }

     iResult = copyfile ( argv[1], argv[2] );
     // ... error messages here
 }
```


Znakowe we-wy

– przykład: modyfikacja programu zliczającego znaki aby zliczał słowa w pliku

```
// counts words in a file
#include <stdio.h>
#include <stdlib.h>                              // for exit()

int main( int argc, char *argv[] )
   {
   FILE *pFile;
   char ch;
   int white = 1;                                // whitespace flag
   int count = 0;                                // word count

   if( argc != 2 )
      { printf("\nUsage: ./program filename");
        exit(1); }

   if( (pFile = fopen(argv[1], "r")) == NULL)
      { printf("\nCan't open file %s.", argv[1]);
        exit(1); }

   while( (ch = getc(pFile)) != EOF )
      {
       switch( ch )
         {
           case ' ' :                           // if space, tab, or
           case '\t':                           // newline, set flag
           case '\n':
              white++; break;
           default:                             // non-whitespace, and
              if(white) { white = 0; count++; } // flag set,
         }                                      // count word
      }
   fclose( pFile );
   printf("\nFile %s contains %d words.", argv[1], count);
   return(0); }
```

Znakowe we-wy

– przykład: dokonaj konwersji pliku z konwencji WIN do konwencji UNIX

```
void main(int argc, char *argv[]) {
char str[25];  char *dot;
int b; int len;
FILE *pOut, *pIn;

if (argc >= 3)
  {
    FILE *pIn  = fopen(argv[1], "rb");
    FILE *pOut = fopen(argv[2], "wb");

    if ( !pIn )
      printf("File %s does not exist e\n",argv[1]);
    else
    {
      printf("Processing of a file:  %s\n" ,argv[1]);

      b = getc( pIn );
      while ( b != EOF )
      {
        if ( b != 13 ) putc(b, pOut);
        b = getc( pIn );
      }
    }
    fclose(pIn);
    fclose(pOut);
   }
else {
    printf("\nToo few parameters\nCalling :\n");
    printf("dos2unix <source file> <destination file>");
} }
```


Znakowe we-wy

– przykład: dokonaj konwersji pliku z konwencji Unix do konwencji WIN

```
void main(int argc, char *argv[]) {
char str[25]; char *dot;     int b, len;    FILE *pOut, *pIn;

if (argc >= 2)
{   pIn = fopen(argv[1], "rb");

    // prepare an output file
    len = strlen( argv[1]) - strlen( dot = strstr(argv[1], "."));
    printf("The substring is: %s\n", dot);
    strncpy(str, argv[1], len); str[len+1] = '\0';  strcat(str, ".txt");

    pOut = fopen(str, "wb");

    if ( !pIn ) printf("File %s does not exist e\n",argv[1]);
    else {
      printf("Processing of a file:  %s\n" ,argv[1]);
      printf("Output file: %s\n" ,str);
      b = getc( pIn );
      while ( b != EOF )
      {
        if (b == '\xa') putc( '\xd', pOut ); // lub '\n' '\r'
        putc( b, pOut ); b = getc( pIn );
      }
    }
   fclose(pIn); fclose(pOut); }
else
{
  printf("\nToo few parameters\nCalling :\n");
  printf("unix2dos <source file> <destination file>");
}                                                          }
```

Znakowe we-wy

– przykład: zmień w pliku char1 na char2 (fragment)

```
int main (int argc, char* argv[]) // .prog IFile, OFile char1 char2
{
char char1, char2;
long int count = 0;
...

char1 = *argv[3]                  // strlen(argv[3]) must not be >1
char2 = *argv[4]                  // strlen(argv[4]) must not be >1


while ( (c = fgetc(pIFile)) != EOF)
{
  if ( c == char1 ) {
    count++;
    c = char2;
  }
  if ( fputc(c, pOFile) == EOF) {
    {
     printf("Error in writing to a file %s", argv[2]);
     return (-1);
    }
}
...
fclose( pIFile);
fclose( pOFile);
...
printf("In file %s, %ld characters %c found\n", argv[1], count, char1);

return 0;
}
```

Jako ćwiczenie: zmodyfikuj program aby można było nim zmieniać jednorazowo większą ilość fraz w pliku.
Przykład wywołania:
./program fraza11 fraza12 fraza 21 fraza 22 **fraza 31 fraza32**

Znakowe we-wy

– przykład: zlicz częstość występowania znaków na wejściu / w pliku

```
#include <stdio.h>
#include <ctype.h>                // for isprint()
#include <limits.h>

unsigned long count[UCHAR_MAX+1];

/* freq main: display byte frequency counts */
int main(void)
{
  int c;

  while ((c = getchar()) != EOF)
    count[c]++;

  for (c = 0; c <= UCHAR_MAX; c++)
    if (count[c] != 0)
      printf("%.2x  %c  %lu\n", c, isprint(c) ? c : '-', count[c]);

 return 0;
}
```

lub ...getc (stdin)...

1. Uruchom program gdy jest EOF
2. zamiast EOF wstaw 'q' i uruchom program

dla danych:

abcde**q**

abc**q**de**q**

– zwróć uwagę na sposób zapamiętania "licznika" każdego znaku w tablicy
`count[]`

– Inny sposób uruchomienia:
`c:\Temp>Project2.exe < main.c`

Znakowe we-wy

– przykład: zlicz częstość występowania znaków na wejściu / w pliku

```
#include <stdio.h>
#include <ctype.h>                  // for isprint()
#include <limits.h>

unsigned long count[UCHAR_MAX+1];

/* freq main: display byte frequency counts */
int main(void)
{
int c;
FILE *fp = fopen("plik", "rb");

while ((c = getc(fp)) != EOF)
  count[c]++;

  for (c = 0; c <= UCHAR_MAX; c++)
    if (count[c] != 0)
      printf("%.2x  %c  %lu\n", c, isprint(c) ? c : '-', count[c]);

 return 0;
}
```

Zamień "rb" na "rt" i zaobserwuj różnice (w systemie Windows)

Znakowe we-wy

– przykład: zlicz częstość występowania znaków na wejściu / w pliku

```
c:\TEMP>Project1.exe
0a  -  5
0d  -  5
20     4
27  '  4
30  0  3
31  1  2
33  3  1
5c  \  2
61  a  6
62  b  6
64  d  1
6e  n  1
72  r  1
78  x  2
```

FILE *fp = fopen("plik", "rb");

```
c:\TEMP>Project1.exe
0a  -  5
20     4
27  '  4
30  0  3
31  1  2
33  3  1
5c  \  2
61  a  6
62  b  6
64  d  1
6e  n  1
72  r  1
78  x  2
```

FILE *fp = fopen("plik", "rt");