argumenty wiersza poleceń: getopt

```
#include <stdio.h>   /*   getopt2.c    getopt2.c  */
#include <stdlib.h>
#include <unistd.h>
  int main(int argc, char *argv[])
{
               int n;
               int opt;

               printf("\n argc=%d\n",argc);
               for(n=0;n<argc;++n)
               {
                printf(" *argv[%d] %s\n",n, argv[n]);
               }
               printf("\n");
               while( (opt=getopt(argc,argv,"a:b:c:")) != -1 )
               {
                switch (opt)
                {
                 case 'a' :
                 printf("case \'a\'\n");
                 printf("   optarg=%s=\n",optarg);
                 break;
                 case 'b' :
                 printf("case \'b\'\n");
                 printf("   optarg=%s=\n",optarg);
                 break;
                 case 'c' :
                 printf("case \'c\'\n");
                 printf("   optarg=%s=\n",optarg);
                 break;
                }

               }

} /* koniec main */
```

Plik (angielski termin: file) to zbiór powiązanych ze sobą danych. Pojęcie pierwotne i zapewne dlatego (również w informatyce) nieprecyzyjnie zdefiniowane.

Dla pracujących na komputerach o strukturze bajtowej pamięci, plik to ciąg bajtów.

Gdzie plik może być przechowywany (może istnieć);

pamięć operacyjna, dyski magnetyczne, taśmy magnetyczne, terminal, tasiemka papierowa, karty perforowane, dysk optyczny (CD) ; ogólnie **nośniki**

nośniki trwałe, nośniki chwilowe

czy ma sens program, który podczas wykonania nie modyfikuje plików ani nie tworzy nowych?

mało prawdopodobne...by taki program był przydatny

Skoro tak, to program powinien mieć możliwość zapisywania/modyfikowania plików, również w celu wymiany ich z innymi programami

instrukcje wejścia/wyjścia

buforowane operacje wyjścia/wejścia WE/WY

(czym się ryzykuje w razie buforowania WE/WY ?)

niebuforowane operacje wyjścia/wejścia

(każde wywołanie WE/WY stanowi obciążenie;
buforowanie minimalizuje to obciążenie)

Ważne: w <stdio.h> są deklaracje struktur i funkcje do wykonywania operacji na plikach.

...rozpoczniemy od przeglądu buforowanych operacji wyjścia/wejścia....

Otwarcie pliku – przy użyciu funkcji **fopen**

```
#include <stdio.h>
FILE *  wsk;  /* wskaźnik do obiektu typu FILE   */
....              /* nazywany również   zmienną plikową  */
.....
wsk = fopen( nazwa,   tryb_otwarcia);
```

```
”nowy.dat” ;                    fopen(”nowy.dat”,”r”);
char buf[]=”nowy.dat”;      fopen(buf,”r”);
```

tryb_otwarcia: read write binary

”r” otwarcie tylko do czytania

”w” otwarcie tylko do pisania; jeśli plik istnieje zostanie obcięty do długości zero

”a” otwarcie do dopisywania (append) ; jeśli zbioru nie ma, zostanie utworzony

”r+” otwarcie istniejącego zbioru do czytania i pisania

(ustawienie na początek pliku)

”w+” otwarcie do czytania i pisania; jeśli plik istnieje zostanie obcięty do pliku o długości zero; jeśli nie istnieje, zostanie utworzony

”a+” otwarcie lub utworzenie pliku do czytania i pisania; jeśli plik istnieje, jego początkowa zawartość nie jest modyfikowana; początkową pozycją do czytania jest początek pliku, zapisywane jest na końcu pliku; jeśli pliku nie ma to zostanie utworzony

„”b” plik binarny (dla systemu UNIX/LINUX w zasadzie jest to bez znaczenia)

instrukcje wejścia/wyjścia **fopen**

”x” istnieje dodatkowo w gcc, zgłasza żądanie by zbiór był utworzony na nowo

możliwe kombinacje ”rb” ”wx” ”wb+”

W pliku stdio.h jest zdefiniowana stała symboliczna EOF, używana jako znacznik błędu albo końca pliku.

W systemach UNIX/LINUX zewnętrzna zmienna całkowita **errno** informuje o przyczynie błędu (aby ta informacja była dostępna, trzeba dołączyć #include <errno.h>)

instrukcje wejścia/wyjścia
clearerr

zeruje znacznik błędu

FILE * wsk;

int clearerr (wsk);

funkcja ta zeruje także znacznik końca pliku

instrukcje wejścia/wyjścia
fclose

zamyka plik

FILE * wsk;

int fclose(wsk);

Daje w wyniku: EOF, gdy wystąpił błąd lub 0 w przeciwnym przypadku

instrukcje wejścia/wyjścia
feof

daje w wyniku TRUE, gdy napotkano koniec pliku

```
FILE * wsk;

int  feof(wsk);
```

przykład feof()

```
#include <stdio.h>

#include <stdlib.h>

  int main()

  {

    FILE * fp1;

    int n=0;


    fp1 = fopen("1.dat","r");
```

przykład feof()

```
while( !feof(fp1) )
    { char c;
      c=getc(fp1);
      printf("%c %d   feof=%d\n", c,c,feof(fp1) );
      /*    clearerr(fp1);   */
    }


  } /*  koniec  funkcji  main  */
```

instrukcje wejścia/wyjścia
ferror

daje w wyniku TRUE, gdy wystąpił błąd operacji na pliku

```
FILE * wsk;

int  ferror(wsk);
```

(chodzi o ostatnią operację na pliku, niezależnie od tego kiedy była wykonana)

uwaga: koniec pliku nie jest błędem operacji na pliku !

przykład ferror()

```
#include <stdio.h>
#include <stdlib.h>
  int main()
  {
    FILE * fp1;
    int n=0;
    fp1 = fopen("1.dat","r");
```

przykład ferror()

```
 while( !feof(fp1) )
 {  char c;
    c=getc(fp1);
    printf("%c %d   feof=%d ferror=%d\n",
                        c, c, feof(fp1), ferror(fp1) );
   /*  clearerr(fp1);  */
 }


} /*  koniec  funkcji  main  */
```

instrukcje wejścia/wyjścia
fflush

wymusza zapisanie bufora

```
FILE * wsk;

int  fflush(wsk);
```

Daje w wyniku EOF, gdy wystąpił błąd lub 0 w przeciwnym razie. Funkcja fflush zapewnia, że dane są rzeczywiście zapisane do pliku. Funkcje wyjścia nie piszą do pliku, lecz tylko umieszczają dane w buforze. Nie można zakładać, że w momencie ”kraksy” zadania bufor zostanie zapisany do pliku tylko dlatego, że użyto np. funkcji fputc.

instrukcje wejścia/wyjścia
fgetc

czyta kolejny znak z pliku

FILE * wsk;

int fgetc (wsk);

Daje w wyniku kod wczytanego znaku albo EOF, jeżeli wystąpił błąd lub koniec pliku.

Uwaga: fgetc zwraca typ int, a nie typ char !!

instrukcje wejścia/wyjścia
fgets

czyta znaki z pliku

FILE * wsk; char * ciąg; int n;

char * fgets(ciąg, n, wsk);

Zwraca adres ciągu przy poprawnym wykonaniu lub NULL, gdy np.. pierwszym odczytanym znakiem jest koniec pliku. Przestaje działać po odczytaniu n-1 znaków lub po napotkaniu ‘\n’ lub EOF. Znak nowego wiersza ‘\n’ jest zapamiętywany w ciągu. Po ostatnim odczytanym znaku jest dodawany znak końca łancucha znakowego ‘\0’ .

instrukcje wejścia/wyjścia
fileno

Daje w wyniku deskryptor pliku.

FILE * wsk;

int fileno (wsk);

Całkowitoliczbowe deskryptory plików są używane tylko w niebuforowanych funkcjach WE/WY.

fileno jest przydatne, gdy do tego samego pliku chcemy po jego jednorazowym otwarciu pisać i w sposób buforowany, i w sposób niebuforowany

instrukcje wejścia/wyjścia **fileno**

```
#include <stdio.h>

#include <stdlib.h>

FILE * fp;

     int  main( )

{ char buf[10]="Karakorum";     int n;

  fp=fopen("nic.dat","w+");    if(fp==NULL) printf("Blad otwarcia
!!\n");

  fprintf(fp,"Eureka...\n");        fprintf(fp,"Eureka2...\n");    fflush(fp);

  printf("\n fileno = %d\n", fileno(fp) );

  write( fileno(fp) , buf , 7);  /* niebuforowana instrukcja WY */

  fprintf(fp,"\n");   exit(0);

}                                    /*   koniec  main   */
```

instrukcje wejścia/wyjścia **fileno**

poniżej zawartość utworzonego zbioru nic .dat

Eureka...

Eureka2...

Karakor

ale gdyby nie było fflush( ), to byłoby

KarakorEureka...

Eureka2...

czyli zafałszowana kolejność zapisów do pliku nic.dat

instrukcje wejścia/wyjścia **fileno**

```
/*  dygresja:  można inaczej  uzyskać deskryptor  */
/*    bez korzystania z funkcji fileno( )          */
  FILE * fp;

  printf("\n fileno = %d\n", fileno(fp) );   /*  lub   */
  printf("\n fileno = %d\n", fp->_fileno );

/* struktura FILE ma pole  _fileno   */
```

instrukcje wejścia/wyjścia
fprintf

Zapisuje argumenty do pliku

FILE *wsk; char * format;

int fprintf(wsk, format, argument,...);

Funkcja fprintf używa tego samego sposobu formatowania co printf i sprintf.

printf pisze do pliku stdout, fprintf do zadanego pliku, a sprintf do łańcucha znakowego (tworzy ciąg znaków)

instrukcje wejścia/wyjścia
fputc

zapisuje znak do pliku

FILE * wsk; char   c;

int fputc(c, wsk);

Jako wynik zwraca 0 przy poprawnym działaniu lub EOF, gdy wystąpił błąd. Parametr fputc jst typu char, podczas gdy analogiczna funkcja wejścia fgetc daje w wyniku wczytany znak w postaci liczby całkowitej typu int.

fputs

Zapisuje ciąg znaków do pliku.

FILE *wsk; char * ciąg;

int fputs(ciąg,wsk);

Wartość zwracana to 0 przy poprawnym zadziałaniu, gdy wystąpił błąd to zwracana jest wartość EOF.

instrukcje wejścia/wyjścia
fread

czyta bloki danych z pliku do tablicy znaków (bezformatowo binarnie)

FILE * wsk; int n1, n2; void * ciąg;

int fread(ciąg, n1,n2,wsk);

Czyta n1*n2 bajtów. Zalecane jest by używać fread z n1=1. Wartość zwracana: liczba wczytanych bloków n2 bądź EOF.

freopen

Zamyka plik i otwiera inny.

FILE * wsk; char * nazwa_pliku; char * tryb_otwarcia;

FILE * freopen(nazwa_pliku, tryb_otwarcia, wsk);

Daje w wyniku EOF, gdy wystąpił błąd.

freopen zamyka plik skojarzony ze wskaźnikiem wsk i otwiera plik o nazwie nazwa_pliku.

instrukcje wejścia/wyjścia
fscanf

Czyta argumenty z pliku.

FILE * wsk; char * format;

int fscanf(wsk, format, argument,....);

Wartość zwracano to liczba odczytanych argumentów lub EOF, gdy wystąpił błąd.

fscanf czyta z zadanego pliku

scanf czyta z pliku stdin

sscanf czyta z łańcucha znakowego (przetwarza ciąg znaków)

instrukcje wejścia/wyjścia
fseek

Zmienia bieżącą pozycję pliku.

FILE * wsk; int odstęp; int początek;

int fseek(wsk, odstęp, początek);

Wartość zwracana: 0 przy poprawnym działaniu, inna liczba gdy wystąpił błąd. Oto jak ustawia pozycję pliku:

początek	bieżąca pozycja ustawiona na
0	odstęp bajtów
1	bieżąca pozycja + odstęp bajtów
2	koniec pliku + odstęp bajtów

instrukcje wejścia/wyjścia
ftell

Zwraca przesunięcie w bajtach od początku pliku do bieżącej pozycji.

```
FILE  * wsk;

long  ftell(wsk);
```

Jeśli wystąpił błąd, to wartościa zwracaną jest -1L .

instrukcje wejścia/wyjścia
fwrite

Zapisuje bloki danych do pliku (działa odwrotnie do fread,

oczywiście jest funkcją bezformatową binarną)

FILE *wsk; void * ciąg; int n1; int n2;

int fwrite(ciag, n1, n2, wsk);

/* n1 to liczba bajtów w bloku, n2 to liczba bloków */

fwrite zapisuje n2 bloków danych z ciągu do pliku wskazywanego przez wsk. Powinna zwrócić n2; każda inna wartość oznacza błąd. Zalecane jest używanie z n1=1 .

instrukcje wejścia/wyjścia
getc

Czyta kolejny znak z pliku.

```
FILE  *  wsk;

int  getc(wsk);
```

Daje w wyniku kod wczytanego znaki (ASCII) albo EOF, gdy wystąpił błąd lub koniec pliku.

instrukcje wejścia/wyjścia
getchar

czyta kolejny znak z pliku stdin

int getchar();

Wartość zwracana to kod wczytanego znaku albo EOF, gdy wystąpił błąd, np. koniec pliku.

gets

czyta znaki z pliku stdin do ciągu znaków (czyta łańcuch znakowy)

char * gets(char * ciąg);

Wartość zwracana to wskaźnik do łańcucha znakowego albo NULL, gdy wystąpił błąd, np. gdy pierwszym wczytanym znakiem jest znak końca pliku.

uwagi: czyta do najbliższego znaku ‘\n’, lecz w przeciwieństwie do fgets() nie przepisuje go !

getw

czyta słowo (tzn. liczbę całkowitą) z pliku

```
FILE * wsk;

int getw(wsk);
```

Zwraca wczytaną liczbę całkowitą lub EOF, gdy wystąpił błąd. Ze względu na to, że EOF jest dopuszczalną liczbą całkowitą, należy użyć feof i ferror, aby sprawdzić czy nastąpił bląd.

instrukcje wejścia/wyjścia
printf

zapisuje argumenty do pliku stdout

char * format;

int printf(format, argument,...);

Wartość zwracana to 0 przy poprawnym działaniu lub EOF, gdy wystąpił błąd.

/* fprintf jest podobna do printf

fprintf(stdout,format,argument,...); */

putc

Zapisuje znak do pliku.

FILE * wsk;

char c;

int putc(c, wsk);

Daje w wyniku kod zapisanego znaku lub EOF, gdy wystąpił błąd.

Zapisuje znak do pliku stdout

char c;

int putchar(c);

Wartość zwracana to kod zapisanego znaku lub EOF, gdy wystąpił błąd.

instrukcje wejścia/wyjścia
puts

Zapisuje ciąg znaków do pliku stdout, kończąc znakiem ‘\n’ .

char * ciąg;

int puts(ciąg);

Zwraca EOF, gdy nastąpi błąd.

puts jest wygodną metodą wypisywania komunikatów do pliku stdout.

```
puts ("This is a message.");
```

putw

zapisuje słowo (liczbę całkowitą) do pliku

```
FILE * wsk;

int  i;

int  putw(i,wsk);
```

Wartość zwracana to zapisana liczba całkowita lub EOF, gdy wystąpił błąd. Ponieważ EOF jest dopuszczalną liczbą całkowitą, należy użyć feof i ferror, aby sprawdzić czy wystąpił błąd.

instrukcje wejścia/wyjścia
scanf

Czyta argumenty z pliku stdin.

char * format;

int scanf(format, argument...);

Wartość zwracana to liczba odczytanych argumentów lub EOF w przypadku błędu.

sscanf

Czyta argument (argumenty) z ciągu znaków.

char * ciąg; char * format;

int sscanf(ciąg, format, argument...);

Wartość zwracana to liczba odczytanych argumentów albo w przypadku błędu EOF.

Funkcja sscanf używa tego samego mechanizmu formatowania co fscanf i scanf. Różnica między tymi funkcjami polega na tym, że scanf czyta z pliku stdin, fscanf z zadanego pliku, a sscanf przetwarza ciąg znaków

tmpfile

Tworzy plik tymczasowy i otwiera go do zapisywania

FILE * tmpfile();

Wartość zwracana to wskaźnik do FILE lub wskaźnik zerowy NULL, gdy wystąpił błąd. Plik tymczasowy jest usuwany przy zakończeniu programu.

```
FILE * wsk;     wsk=tmpfile() ;

int rak=7;

putw(rak,wsk);

fseek(wsk,0,0);
```

ungetc

Powoduje wycofanie znaku do pliku, tak że następne wykonanie getc da ponownie ten sam znak.

```
FILE * wsk;

int c;

int ungetc(c, wsk);
```

Wartość zwracana to znak c (jako wartość ASCII) lub EOF, gdy wystąpił błąd. Tylko jeden znak może być wycofany bez ponownego odczytu. EOF nie może być wycofany. Funkcja fseek anuluje efekty działania ungetc.

unlink

usuwa plik

```
#include <unistd.h>                /*  tam jest prototyp funkcji  */

int  unlink(char * nazwa_pliku);
```

Wartością zwracaną jest 0 lub EOF, jeśli nastąpił błąd.

Parametrem funkcji jest nazwa pliku, a nie wskaźnik do FILE.

Oznacza to, że nie jest konieczne otwarcie pliku przed jego usunięciem.

rmdir

usuwa kartotekę

```
#include <unistd.h>                    /*  tam jest prototyp funkcji  */

int  rmdir(char * nazwa_pliku);
```

Wartością zwracaną jest 0 lub EOF, jeśli nastąpił błąd.

Parametrem funkcji jest nazwa kartoteki, a nie wskaźnik do FILE.

Kartoteka musi być pusta by mogła być usunięta.

remove

usuwa plik lub kartotekę

```
#include <stdlib.h>                /*  tam jest prototyp funkcji  */

int  remove(char * nazwa);
```

Wartością zwracaną jest 0 lub EOF, jeśli nastąpił błąd.

Działa jak unlink dla zbiorów i jak rmdir dla kartotek.

instrukcje wejścia/wyjścia

rename

```
int rename( char *old,  char *new);
```

Funkcja zwraca wartość -1, jeśli będzie błąd. Dodatkowo ustawia kod błędu w zmiennej **errno**, możliwe wartości to np.

ENOENT informujący że plik nie istnieje, czy **EACCES** informujący że nie ma zezwolenia na dostęp do pliku

instrukcje wejścia/wyjścia niebuforowane

close creat eof lseek

open read tell write

(jest ich znacznie mniej niż buforowanych)

instrukcje wejścia/wyjścia niebuforowane

open

```
/*   open  otwiera  zbiór  */

#include <fcntl.h>

#include <sys/type.h>

#include <sys/stat.h>

int open (const char *filename, int flags[, mode_t mode])


open(”22.dat”,  O_RDONLY,  0664);

    /* trzeci parametr ma znaczenie w przypadku tworzenia

        pliku  - kod dostępu dla użytkownika, grupy, świata */
```

instrukcje wejścia/wyjścia niebuforowane

open

O_RDONLY otwarty do odczytu

O_WRONLY otwarty do zapisu

O_RDWR otwarty do odczytu i zapisu

O_APPEND dodaje nowe dane na końcu pliku

O_CREAT tworzy plik

O_TRUNC jeśli plik istnieje, jest obcinany do długości zero

O_EXCL istnienie pliku to błąd

O_BINARY otwarty w trybie binarnym (tylko w systemie WINDOWS)

O_TEXT otwarty w trybie tekstowym (tylko w systemie WINDOWS)

instrukcje wejścia/wyjścia niebuforowane

open

O_EXLOCK program dostaje wyłączność na używanie pliku

(”exclusive lock”)

/* w przypadku błędu open zwraca wartość –1 */

/* dodatkowo następujące warunki błędu dostępne przez ***errno*** są dostarczane : */

open

EACCES plik istnieje lecz nie można go czytać/pisać po nim jak zażądano we *flags* ; plik nie istnieje i nie ma zezwolenia na zapis w danej kartotece tak więc plik nie może być utworzony

EEXIST jednocześnie O_CREAT oraz O_EXCL są wymienione we *flags* zaś wymieniony zbiór już istnieje

EINTR operacja open została przerwana przez zewnętrzny sygnał

EISDIR argument we *flags* wymaga ”write access” zaś plik jest kartoteką

EMFILE zadanie (proces) ma za dużo otwartych plików

ENFILE system zbiorów odmawia zezwolenia na otwarcie kolejnego pliku

open

ENOENT plik o podanej nazwie nie istnieje, zaś we *flags* nie wymieniono O_CREAT

ENOSPC nie można utworzyć nowego pliku, gdyż nie ma miejsca na urządzeniu **(”no disk space left”)**

instrukcje wejścia/wyjścia niebuforowane **open**

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
 #include <sys/types.h>
 #include <sys/stat.h>
     int main ()
{ char buf[20]="Kaskada";     int index;  /* to będzie deskryptor  pliku*/
 index=open("nic.dat",O_RDWR|O_TRUNC|O_CREAT, 0664);
/* do czytania i pisania; jeśli istnieje skróć plik do zera;
  jeśli plik nie istnieje to go utwórz  */
 *(buf+7)='\n';
 write(index,buf,8);    exit(0);      }      /*   koniec  main   */
```

instrukcje wejścia/wyjścia niebuforowane **open**

/* jaka będzie zawartość zbioru nic.dat po wykonaniu programu
z poprzedniego slajdu ? */

instrukcje wejścia/wyjścia niebuforowane **open**
S.Oualline, rozdział 14, strona 246

pisanie/czytanie buforowane i niebuforowane.....

(deskryptor)	*(nazwa symboliczna)*	*(przeznaczenie pliku)*
0	STDIN_FILENO	standardowe wejście
1	STDOUT_FILENO	standardowe wyjście
2	STDERR_FILENO	standardowa obsługa błędów

instrukcje wejścia/wyjścia niebuforowane **creat**

/* tworzy plik */

int **creat** *(const char *filename, mode_t mode)*

ta postać jest zastępowana przez

open (*filename*, O_WRONLY | O_CREAT | O_TRUNC, *mode*)

/* co to jest mode ? */

instrukcje wejścia/wyjścia niebuforowane **close**

/* zamyka plik */

/* może być konieczne dołączenie #include <unistd.h> */

int **close** (int deskryptor_pliku);

Zwraca wartość –1, gdy wystąpił błąd. Otwarte pliki są zamykane automatycznie w momencie zakończenia programu. Jednak ze względu na możliwość załamania się zadania (systemu) w najmniej odpowiedniej chwili, zaleca się zamykania plików, gdy przestają być wykorzystywane.

instrukcje wejścia/wyjścia niebuforowane **eof**

/* sprawdza znacznik końca pliku */

int **eof** (deskryptor_pliku);

Wartość zwracana to –1, gdy wystąpił błąd funkcji **eof, 0** gdy nie ma końca pliku oraz **1**, gdy wystąpił koniec pliku.

uwaga: funkcji eof() często nie ma (np. w gcc)

instrukcje wejścia/wyjścia niebuforowane **lseek**

```
/*  zmienia bieżącą pozycję pliku     -   jak  fseek  dla  buforowanych*/
#include <unistd.h>
long  lseek (int    deskryptor pliku,    int odstęp,    int początek);
```

Wartością zwracaną jest **-1L**, gdy wystąpił błąd.

początek	bieżąca pozycja ustawiona na
0	odstęp bajtów
1	bieżąca pozycja + odstęp bajtów
2	koniec pliku + odstęp bajtów

instrukcje wejścia/wyjścia niebuforowane **tell**

/* podaje bieżącą pozycję w pliku - w bajtach */

long **tell** (int deskryptor_pliku);

Daje w wyniku –1, gdy nastąpi błąd

instrukcje wejścia/wyjścia niebuforowane **read**

#include <unistd.h>

int **read** (int deskryptor_pliku, void *buffer, int size) ;

buffer to wskaźnik do miejsca pamięci, gdzie wczytany ciąg bajtów jest zapisywany; **uwaga –** ciąg bajtów nie jest uzupełniany bajtem zerowym!

size – ile bajtów wczytać; jeśli nie ma tyle bajtów w zbiorze to wczytanych zostanie odpowiednio mniej

Wartość zwracana - liczba przeczytanych bajtów lub –1, gdy nastąpił błąd.

instrukcje wejścia/wyjścia niebuforowane **write**

```
#include  <unistd.h>

int  write  (int deskryptor_pliku,   void *buffer, int   size) ;
```

buffer to wskaźnik do miejsca pamięci skąd pobierany jest ciąg bajtów do zapisania w pliku o deskryptorze **deskryptor_pliku;** ten ciąg bajtów nie jest uzupełniany bajtem zerowym !

Wartość zwracana – liczba zapisanych bajtów lub –1, gdy wystąpił błąd.

FILE * **fdopen** *(int deskryptor, const char *tryb_otwarcia)*

fdopen zwraca wskaźnik do FILE, umożliwia zatem dalsze pisanie po pliku w sposób buforowany (gdy wcześniej otwarto plik w sposób niebuforowany przez **open** i uzyskano deskryptor). Argument *tryb_otwarcia* jest taki sam jak w przypadku **fopen**, np. ”w” czy ”w+” .

(poznalismy już **fileno** które działa ”w przeciwnym kierunku”)

pożyteczne funkcje pracujące na pojedynczych znakach

#include <ctype.h>

Testowanie znaku:

int **isalnum**(int c) – prawda jeśli c jest alphanumeryczne
int **isalpha**(int c) – prawda jeśli c jest znakiem
int **isascii**(int c) – prawda jeśli c jest znakiem ASCII
int **iscntrl**(int c) – prawda jeśli c jest znakiem kontrolnym
int **isdigit**(int c) – prawda jeśli c jest cyfrą
int **isgraph**(int c) – prawda jeśli c jest znakiem graficznym
int **islower**(int c) – prawda jeśli c jest małą literą

pożyteczne funkcje pracujące na pojedynczych znakach

#include <ctype.h>

Testowanie znaku:

int **isprint**(int c) – prawda jeśli c jest znakiem drukowalnym
int **ispunct** (int c) – prawda jeśli c jest znakiem interpunkcyjnym, tj. jest znakiem drukowalnym, nie alfanumeryczny ani spacja

int **isspace**(int c) – prawda jeśli c jest znakiem spacja
int **isupper**(int c) – prawda jeśli c jest dużą literą
int **isxdigit**(int c) – prawda jeśli c jest cyfrą heksadecymalną

pożyteczne funkcje pracujące na pojedynczych znakach

#include <ctype.h>

Konwersja znaku:

int toascii(int c) – zamienia c na ASCII, poprzez ustawienie bitów nr 8,9,... na zero

int tolower(int c) – zamienia c na małą literę

int toupper(int c) – zamienia c na dużą literę

errno – **kod błędu**

Większość funkcji bibliotecznych zwraca szczególną wartość dla zaznaczenia, że zawiodły. np. –1, zerowy wskaźnik NULL, stałą jak EOF itp.. Mówi to jednak tylko, że zdażył się błąd. Jaki? należy skontrolować jaki **kod błędu** został zapisany w errno. Zmienna erno jest zadeklarowana w errno.h . Kody błędu są przedstawione w opisach funkcji.

Funkcje nie zmieniają wartości errno jeśli błędu nie było. Tym samym wartość errno po udanym zawołaniu funkcji niekoniecznie jest zero!

Oczywiście po tym jak funkcja zakończyła się błędnie, można sprawdzić, co zapisane jest w errno; można także wyzerować errno przed zawołaniem funkcji.

errno

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int n;
    main()
{                   /*     rmdir("testowa_kartoteka");   */
     n=mkdir("testowa_kartoteka",0774);
     if(errno==EEXIST) printf("\n  !! EEXIST=%d\n",EEXIST);
   printf("\n n=%d\n",n);
   exit(0);  }                    /* koniec   funkcji  main*/
```

funkcje matematyczne

należy dołączyć plik **#include <math.h>** oraz w kompilacji dodać pewien klucz

```
gcc program.c  -lm
```

abs wartość bezwzględna (zwraca wartość typu int)

acos arcus cosinus

asin arcus sinus

atan arcus tangens

funkcje matematyczne

atan2 arcus tangens (pewna odmiana)

atoi przekształca ciąg znaków na liczbę całkowitą

atof przekształca ciąg znaków na liczbę rzeczywistą

atol przekształca ciąg znaków na liczbę całkowitą long

ceil podaje najmniejszą liczbę całkowitą nie mniejszą niż

cos cosinus

cosh cosinus hiperboliczny

exp e do potęgi argument

fabs wartość bezwzględna (zwraca wartość rzeczywistą)

funkcje matematyczne

floor największa liczba całkowita nie większa od argumentu

fmod dzielenie modulo (reszta z dzielenia)

log logarytm naturalny

log10 logarytm dziesiętny

pow potęga pow(a,b) to a do potęgi b

sin sinus

sinh sinus hiperboliczny

sqrt pierwiastek kwadratowy

tan tangens

tanh tangens hiperboliczny

funkcje matematyczne

srand48 - służy do inicjalizacji generatora liczb pseudolosowych z przedziału (0., 1.)

void **srand48** *(long int seed)*

drand48 - generator liczb pseudolosowych z przedziału (0.,1.)

double **drand48** *(void)*

stałe matematyczne w <math.h>

HUGE maksymalna wartość liczby zmiennoprzecinkowej typu float

M_E liczba e

M_LOG2E logarytm o podstawie 2 z liczby e

M_LOG10E logarytm o podstawie 10 z liczby e

M_LN2 logarytm naturalny z 2

M_LN10 logarytm naturalny z 10

M_PI π

stałe matematyczne w <math.h>

M_PI_2 π /2.

M_PI_4 π /4.

M_1_PI 1/ π

M_2_PI 2/ π

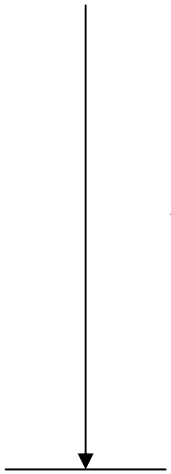
M_2_SQRTPI 2/ sqrt (π)

M_SQRT2 sqrt(2)

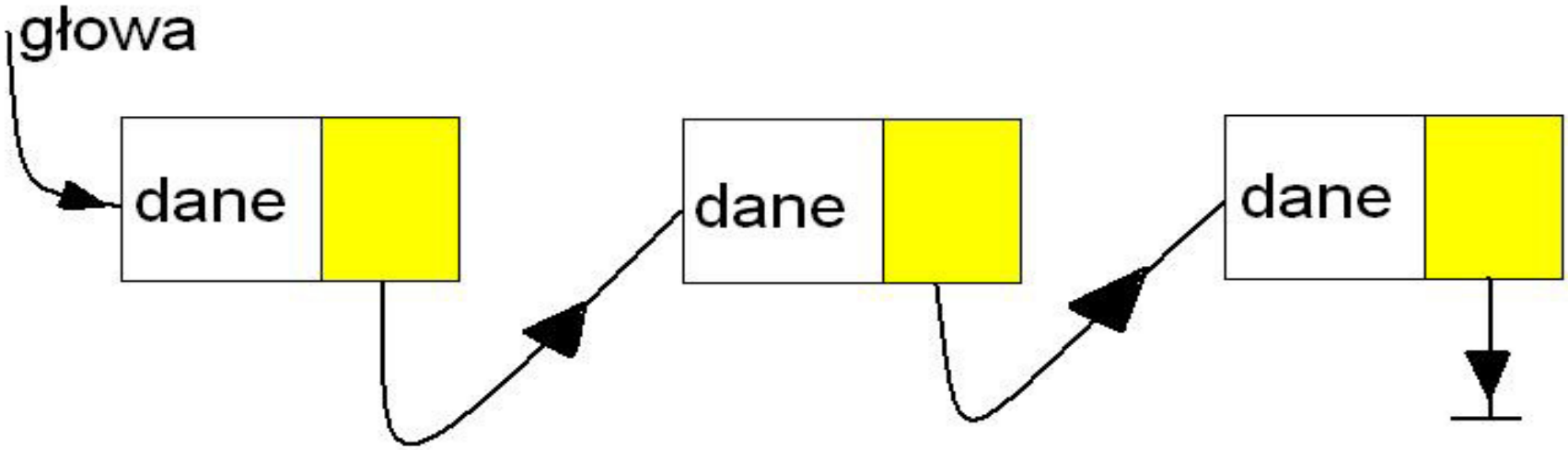
M_SQRT1_2 1./sqrt(2)

MAXFLOAT -- The maximum value of a non-infinite single-precision floating point number.

HUGE_VAL - dodatnia nieskończoność



wskaźnik zerowy NULL



lista jednokierunkowa

odmierzanie odcinków czasu

```
#include <sys/types.h>
#include <sys/times.h>
main()
{ struct tms before, after ;
   times(&before);
                    /* ... place code to be timed here ... */
    times(&after);
    printf("User time: %ld\n", after.tms_utime - before.tms_utime);
    printf("System time: %ld\n", after.tms_stime - before.tms_stime);
/* czas podany w tikach zegara   */
   exit(0);   } /*   koniec funkcji  main    */
```

odmierzanie odcinków czasu

```
#include <time.h>

clock_t  start, end;

double cpu_time_used;

start = clock(); ...

/* tu program coś robi... */

end = clock();

cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

odmierzanie odcinków czasu – funkcja **times**

```
#include <sys/times.h>
        struct tms  alfa;   /* potrzebna definicja obiektu  */
        struct tms *buffer;
        buffer=& alfa;
clock_t times (struct tms *buffer)  /*  wywołanie funkcji    */
```

clock_t tms_utime

Całkowity czas procesora które proces (zadanie) zużyło na wykonywanie instrukcji procesu

clock_t tms_stime

czas który procesor zużył na obsługę procesu

(funkcja **times** umieszcza informację o zużytym czasie w *buffer)

odmierzanie odcinków czasu – funkcja **clock()**

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

clock_t **clock** (*void*)

funkcja ta zwraca bieżący czas CPU; sens ma przede wszystkim różnica wartości zwracanych przez kolejne wywołanie **clock()**

clock_t == int (== long int)

w przypadku błędu **clock()** zwraca wartość (clock_t)(-1)

odmierzanie odcinków czasu – przykład

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

#include <sys/times.h>

#include <math.h>


struct tms  alfa;
```

odmierzanie odcinków czasu – przykład

```
int main()
{
  float sum;
  long n,m,na,nb;
  int k;
  struct tms * buffer;
  buffer= &alfa;
  printf("  CLOCKS_PER_SEC %d\n",
                CLOCKS_PER_SEC);
```

odmierzanie odcinków czasu – przykład

```
na= times(buffer);
m= clock();
printf("\n pocz.  (*buffer).tms_utime %d\n", (*buffer).tms_utime );
printf("\n pocz.  (*buffer).tms_stime %d\n", (*buffer).tms_stime );
printf("\n pocz.  m=%d   clock()\n",m);
sum=0.;
for(k=0;k<1000*1000*10;++k)
  sum+= k/1.e5 + exp(-5.+(float)k/1.375e7)+log( (float)k);
```

odmierzanie odcinków czasu – przykład

```
  nb= times(buffer);
  n = nb - na;
  m= clock();

  printf("\n  (*buffer).tms_utime %d\n", (*buffer).tms_utime );
  printf("\n  (*buffer).tms_stime %d\n", (*buffer).tms_stime );
  printf("\nn=%d      nb=%d   na=%d\n",n, nb, na);
  printf("\n  m=%d   clock()",m);
  printf("\n");

} /* koniec main */
```

odmierzanie odcinków czasu – przykład

CLOCKS_PER_SEC 1000000

pocz. (*buffer).tms_utime 0

pocz. (*buffer).tms_stime 0

pocz. m=469 clock()

(*buffer).tms_utime 67

(*buffer).tms_stime 0

n=67 nb=441236179 na=441236112
m=670770 clock()

czas CPU

(czas kalendarzowy, liczony od jakiegoś zdarzenia)

czas procesora

(czas zużyty, np. na przeczytanie książki zużyto 7 godzin)

(w GNU C wyjątkowo clock() zwraca zużyty „system time + user time”)

Np. time() zwraca czas kalendarzowy liczony w sekundach od 1 stycznia 1970, godz. 00:00.

Czas kalendarzowy w dużej rozdzielczości

Można go uzyskać np. używając funkcji **gettimeofday**

int **gettimeofday** (*struct timeval *tp, NULL*)

```
struct timeval {
        time_t      tv_sec;     /* seconds */
        suseconds_t tv_usec;    /* microseconds */
      };
```