



Politechnika
Wroclawska

Wejście/wyjście i operacje na łańcuchach znaków wer. 10 z drobnymi modyfikacjami!

Wykład 6

Wojciech Myszka

Katedra Mechaniki, Inżynierii Materiałowej i Biomedycznej

2022-05-17 16:59:04 +0200





Politechnika
Wroclawska

Wejście/wyjście — pliki

Wykład 6

Wojciech Myszka

Katedra Mechaniki, Inżynierii Materiałowej i Biomedycznej

17 maja 2022





Część I

Formatowane (tekstowe) wejście/wyjście

Otwarcie pliku I

1. Przed skorzystaniem z pliku należy go „otworzyć” (*open*).
2. **stdin**, **stdout**, **stderr** otwarte są w trybie tekstowym.
3. Standardowa biblioteka (`stdio.h`) zawiera trzy funkcje `fopen`, `freopen` i `fclose`.
4. Pierwsze dwie służą do „skojarzenia” (powiązania) pliku z odpowiednią strukturą danych.
5. Ostatnia — powiązanie to likwiduje.
6. Wszystkie czynności związane z otwieraniem i zamykaniem plików realizowane są przez System Operacyjny.

Operację otwarcia pliku (ponownego otwarcia) realizują funkcje **fopen** i **freopen**:

Otwarcie pliku II

```
#include <stdio.h>  
FILE *fopen(const char *filename, const char *mode);  
FILE *freopen(const char *filename, const char *mode,  
              FILE *stream);
```

Otwarcie pliku III

Użycie:

```
FILE *fp ;  
...  
fp = fopen("plik.txt", "w");  
...  
fp = freopen("plik.txt", "r", fp);
```

- ▶ Pierwszy parametr (filename) to wskaźnik do tablicy tekstowej (na przykład stałej) zawierającej nazwę pliku.
- ▶ Parametr drugi to ciąg znaków (lub zmienna) zawierająca w sposób symboliczny określenie trybu dostępu:
 - ▶ **r** otwórz plik **tekstowy** do odczytu (plik musi istnieć)
 - ▶ **w** otwórz plik **tekstowy** do zapisu (niszcząc jego zawartość jeżeli już istnieje)

Otwarcie pliku IV

- ▶ **a** (*append*); otwórz plik **tekstowy** do dopisywania, jeżeli plik nie istnieje — zostanie utworzony.
- ▶ **r+** otwarcie pliku **tekstowego** do zapisu i odczytu (plik powinien już istnieć!)
- ▶ **w+** otwórz istniejący plik (kasując jego zawartość) lub utwórz nowy plik **tekstowy** w trybie do zapisu i odczytu
- ▶ **a+** otwórz plik **tekstowy** na końcu w trybie do zapisu i odczytu



Otwarcie pliku V

Przykład

```
#include <stdio.h>
...
FILE *pp;
...
pp = fopen("Ala.txt", "r");
if (pp == NULL)
    { /* Błąd! */}
...
fscanf(pp, "%d", &i);
```

- ▶ Trzeci parametr (stream — tylko w przypadku funkcji reopen) zawiera wskaźnik do struktury danych opisujących strumień danych.

Otwarcie pliku VI

1. Funkcja **fopen()** otwiera plik o podanej nazwie we wskazanym trybie, tworzy strukturę danych opisującą go i zwraca do niej adres. W przypadku błędu — zwraca NULL.
2. Funkcja **freopen()** najpierw zamyka otwarty wcześniej plik i otwiera go ponownie we wskazanym trybie, zwracając wskaźnik do struktury danych opisujących strumień. W przypadku błędu — zwraca NULL.
3. Funkcja **fclose()** zamyka wskazany plik. W przypadku błędu — zwraca NULL.

```
#include <stdio.h>  
...  
int fclose(FILE *stream);
```

stream to wskaźnik do struktury danych opisujących strumień danych.
Przykład:

Otwarcie pliku VII

```
...  
fclose( fp );  
...
```

Odczyt formatowany I

1. Funkcje typu `scanf`

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...)
;

int scanf(const char *format, ...);

int sscanf(const char *s, const char *format,
...);
```

Dane czytane są ze wskazanego strumienia (`stream`) i interpretowane zgodnie z użytym formatem. W formacie powinna wystąpić wystarczająca liczba specyfikacji aby dostarczyć dane dla wszystkich argumentów.

Odczyt formatowany II

- ▶ `fscanf` zwraca liczbę przeczytanych wartości lub wartość EOF
- ▶ `scanf` jest odpowiednikiem `fscanf`, ale dotyczy strumienia `stdin`
- ▶ `sscanf` jest odpowiednikiem `fscanf`, z tym, że dane „czytane” są z tablicy znakowej; dotarcie do końca tabeli jest równoważne z dotarciem do końca pliku

Rodzina poleceń fprintf |

Wszystkie funkcje realizują formatowane wyprowadzanie informacji.

```
#include <stdarg.h>
#include <stdio.h>
int fprintf(FILE *stream, const char *format, \
            ...);
int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
int vfprintf(FILE *stream, const char *format, \
             va_list arg);
int vprintf(const char *format, va_list arg);
int vsprintf(char *s, const char *format, \
             va_list arg);
```

- ▶ **fprintf** to podstawowa wersja funkcji

Rodzina poleceń fprintf II

- ▶ **printf** używa stdout jako strumienia wyjściowego.
- ▶ **sprintf** przekazuje sformatowane informacje do tablicy znakowej (odpowiedniej długości — musi zadbać programista)
- ▶ **snprintf** dodatkowy parametr mówi o długości tablicy znakowej, do której mają być zapisywane dane; dłuższe zostaną przycięte!
- ▶ warianty o nazwie rozpoczynającej się na v (vfprintf, fprintf, vsprintf) używają specyficznej metody przekazywania listy parametrów zmiennej długości — nie będziemy się nimi zajmować.

Funkcje pomocnicze I

Poniższe funkcje są nieco mniej istotne (na tym poziomie nauki języka C)

1. fflush()

```
#include <stdio.h>
int fflush(FILE *stream);
```

Funkcja powoduje zapis w pliku (związanym ze strumieniem wskazywanym przez stream) wszystkich zbuforowanych danych.

2. setvbuf()

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, \
            size_t size);
```

- ▶ funkcji można użyć zaraz po otwarciu pliku, ale przed wykonaniem jakiegokolwiek innej operacji na pliku



Funkcje pomocnicze II

- ▶ zmienna `mode` określa sposób buforowania danych (`_IOFBF` — pełne buforowanie, `_IOLBF` — buforowanie linii, `_IONBF` — buforowanie wyłączone)
- ▶ pozostałe parametry (`buf` i `size`) określają — jeżeli `buf` nie jest `NULL` — obszar (i jego wielkość) przeznaczony do buforowania danych. Jeżeli `buf` jest równe `NULL` funkcja przydziela taki obszar automatycznie. Funkcja zwraca zero gdy zakończy się normalnie i wartość różną od zera w przypadku błędów.



Pozycja w pliku I

1. fgetpos() i fsetpos()

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

- ▶ Funkcja **fgetpos** zapisuje aktualną wartość wskaźnika pozycji związanego ze strumieniem stream w obiekcie wskazywanym przez pos
- ▶ Funkcja **fsetpos** ustala aktualną wartość wskaźnika pozycji strumienia stream na wartość z obiektu wskazywanego przez pos. Wartość ta powinna być wcześniej uzyskana za pomocą funkcji fgetpos.



Pozycja w pliku II

2. fseek() i ftell()

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, \
          int whence);
long int ftell(FILE *stream);
```

- ▶ Funkcja **ftell** zwraca aktualną wartość wskaźnika pozycji (dla plików binarnych jest to liczba znaków od początku plików; dla plików tekstowych sprawa jest bardziej skomplikowana) strumienia określonego przez stream.
- ▶ Funkcja **fseek** dla plików binarnych ustala aktualną wartość wskaźnika pozycji przez sumowanie zmiennej offset z pozycją wskazywaną przez whence; stdio.h zawiera makra SEEK_SET, SEEK_CUR i SEEK_END określające odpowiednio początek pliku, bieżącą pozycję i koniec pliku. Dla plików tekstowych wartość offset powinna być albo zero albo wartością uzyskana wcześniej przez odwołanie do funkcji **ftell**.



Pozycja w pliku III

3. `rewind()`

```
#include <stdio.h>  
void rewind(FILE *stream);
```

Funkcja ustawia wskaźnik pozycji w pliku na początek pliku.



Czytanie z pliku I

1. fgetc()

```
#include <stdio.h>  
int fgetc(FILE *stream);
```

Funkcja zwraca następny znak (jako unsigned char zamieniony na int) ze strumienia wskazywanego przez stream. Jeżeli odczyt dotrze do końca pliku funkcja zwraca EOF. W przypadku błędu ustawiany jest wskaźnik błędu i funkcja zwraca EOF.

2. fgets()

```
#include <stdio.h>  
char *fgets(char *s, int n, FILE *stream);
```



Czytanie z pliku II

Funkcja odczytuje co najwyżej $n-1$ znaków z pliku wskazywanego przez stream i zapisuje je w tablicy wskazywanej przez s . Odczyt kończy się z chwilą napotkania znaku nowej linii, dojścia do końca strumienia lub przeczytania $n-1$ znaków.

Uwaga: Znak nowej linii to $\backslash n$ dla Unixa, $\backslash r \backslash n$ dla DOS/Windows i $\backslash r$ dla MAC (przed OS X).

3. **getc()** — odpowiednik **fgetc**, może być zaimplementowana jako makro.
4. **getchar()** — odpowiednik **getc** ale dla **stdin**
5. **ungetc()**

```
#include <stdio.h>  
int ungetc(int c, FILE *stream);
```

Funkcja „zwraca” znak c do wskazanego strumienia. Operacja może się nie powieść, jeżeli zwracamy w ten sposób zbyt wiele znaków bez wykonania operacji czytania lub pozycjonowania pliku. Zawsze działa dla jednego znaku.

Plik wejściowy nie ulega zmianie!



Odczyt formatowany I

1. Funkcje typu scanf

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
int scanf(const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

Dane czytane są ze wskazanego strumienia (stream) i interpretowane zgodnie z użytym formatem. W formacie powinna wystąpić wystarczająca liczba specyfikacji aby dostarczyć dane dla wszystkich argumentów.

- ▶ fscanf zwraca liczbę przeczytanych wartości lub wartość EOF
- ▶ scanf jest odpowiednikiem fscanf, ale dotyczy strumienia stdin
- ▶ sscanf jest odpowiednikiem fscanf, z tym, że dane „czytane” są z tablicy znakowej; dotarcie do końca tabeli jest równoważne z dotarciem do końca pliku





Część II

Binarne wejście/wyjście

„Otwarcie” pliku I

```
#include <stdio.h>
FILE *fopen(const char *filename ,
            const char *mode);
FILE *freopen(const char *filename ,
              const char *mode, FILE *stream );
...
FILE *fp ;
```

- ▶ Pierwszy parametr (filename) to wskaźnik do tablicy tekstowej (na przykład stałej) zawierającej nazwę pliku.
- ▶ Parametr drugi to ciąg znaków (lub zmienna) zawierająca w sposób symboliczny określenie trybu dostępu:
 - ▶ **rb** otwórz plik binarny do czytania

„Otwarcie” pliku II

- ▶ **wb** utwórz plik do zapisu w trybie binarnym, jeżeli plik istnieje, jego zawartość zostanie skasowana
 - ▶ **ab** otwórz plik binarny w trybie do dopisywania (jeżeli plik nie istnieje — zostanie utworzony).
 - ▶ **r+b** lub **rb+** zapis i odczyt dla plików binarnych (plik musi istnieć)
 - ▶ **w+b** lub **wb+** otwórz w trybie binarnym plik istniejący (kasując jego zawartość) lub utwórz plik nowy w trybie do zapisu i odczytu
 - ▶ **a+b** lub **ab+** otwórz plik w trybie binarnym na końcu w trybie do zapisu i odczytu
- ▶ Trzeci parametr (stream — tylko w przypadku funkcji reopen) zawiera wskaźnik do struktury danych opisujących strumień danych.

1. fread()

```
#include <stdio.h>
size_t fread(void *ptr, size_t size,
             size_t nmemb, FILE *stream);
```

Funkcja realizująca dostęp bezpośredni do pliku, pozwala odczytać **nmemb** elementów o wielkości **size** każdy do tablicy wskazywanej przez **ptr** ze strumienia **stream**.

Funkcja zwraca liczbę przeczytanych elementów, która może być mniejsza od **nmemb** gdy wystąpi błąd lub funkcja dojdzie do końca pliku.

2. fwrite()

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

Funkcja zapisuje binarnie elementy tablicy wskazanej przez **ptr** do strumienia **stream**. **size** określa wielkość jednego obiektu, a **nmemb** liczbę obiektów.



Zapis tablicy I

```
#include <stdio.h>
int main()
{
    FILE *pp;
    double t[10] = {
        0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    };
    pp = fopen("Ala.txt", "wb");
    if ( pp == NULL )
    {
        printf("Blad!\n");
        return 1;
    }
}
```

Zapis tablicy II

```
fwrite(t, sizeof( double ), 10, pp);  
fclose(pp);  
return 0;  
}
```



Pliki binarne I

Prosty przykład

```
/*  
 * Odczyt jednej liczby w trybie binarnym  
 */  
#include <stdio.h>  
int main(void)  
{  
    FILE *fp;  
    unsigned int e;  
    fp = fopen("foo.txt", "rb");  
    if ( fp == NULL )  
    {  
        printf("blad1\n");  
        return 1;  
    }  
}
```



Pliki binarne II

Prosty przykład

```
}  
if ( fread(&e, sizeof( e ), 1, fp) == 0 )  
{  
    printf("blad2\n");  
    return 1;  
}  
printf("e_=_%u\n", e);  
return 0;  
}
```

Pliki binarne III

Prosty przykład

1. Plik foo.txt nie istnieje

```
$ ls
```

```
b.c  Debug
```

```
$ Debug/binarne
```

```
blad1
```


Pliki binarne IV

Prosty przykład

2. Plik foo.txt istnieje, ale jest pusty

```
$ touch foo.txt
```

```
$ ls -l
```

```
razem 8
```

```
-rw-r--r-- 1 myszka myszka 319 2008-05-05 12:05 b.c
```

```
drwxr-xr-x 2 myszka myszka 4096 2008-05-05 12:05 Debug
```

```
-rw-r--r-- 1 myszka myszka 0 2008-05-05 12:12 foo.txt
```

```
$ Debug/binarne
```

```
blad2
```

Pliki binarne V

Prosty przykład

3. Plik istnieje, ma długość trzech bajtów:

```
$ ls -l
razem 12
-rw-r--r-- 1 myszka myszka 319 2008-05-05 12:05 b.c
drwxr-xr-x 2 myszka myszka 4096 2008-05-05 12:05 Debug
-rw-r--r-- 1 myszka myszka 3 2008-05-05 12:15 foo.txt
$ Debug/binarne
blad2
```

Pliki binarne VI

Prosty przykład

4. Plik istnieje, ma długość czterech bajtów:

```
$ cat foo.txt
```

```
abc
```

```
$ ls -l
```

```
razem 12
```

```
-rw-r--r-- 1 myszka myszka 319 2008-05-05 12:05 b.c
```

```
drwxr-xr-x 2 myszka myszka 4096 2008-05-05 12:05 Debug
```

```
-rw-r--r-- 1 myszka myszka 4 2008-05-05 12:18 foo.txt
```

```
$ Debug/binarne
```

```
e = 174285409
```



Zapis i odczyt I

```
#include <stdio.h>
int main(void)
{
    FILE *fp;
    unsigned int i, k;
    fp = fopen("foo.txt", "wb");
    if ( fp == NULL )
    {
        printf("blad1\n");
        return 1;
    }
    for ( i = 0; i < 10; i++ )
    {
```



Zapis i odczyt II

```
fseek(fp, i * sizeof( i ), SEEK_SET);
fwrite(&i, sizeof( i ), 1, fp);
}
fclose(fp);
fp = fopen("foo.txt", "rb");
if ( fp == NULL )
{
    printf("blad1\n");
    return 1;
}
for ( i = 0; i < 10; i++ )
{
    fseek(fp, ( 9 - i ) * sizeof( i ), SEEK_SET);
    fread(&k, sizeof( i ), 1, fp);
```

Zapis i odczyt III

```
        printf("k=□%d\n", k);  
    }  
    fclose(fp);  
    return 0;  
}
```

Zapis i odczyt (wariant drugi) I

```
#include <stdio.h>
int main(void)
{
    FILE *fp;
    unsigned int i, k;
    fp = fopen("foo.txt", "w+b");
    if ( fp == NULL )
    {
        printf("blad1\n");
        return 1;
    }
    for ( i = 0; i < 10; i++ )
    {
```

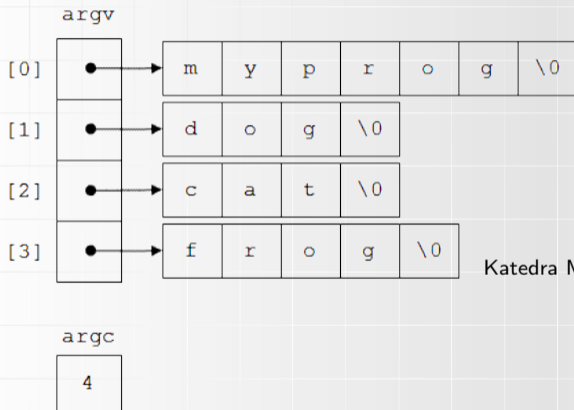


Zapis i odczyt (wariant drugi) II

```
fseek(fp, i * sizeof( i ), SEEK_SET);  
fwrite(&i, sizeof( i ), 1, fp);  
}  
for ( i = 0; i < 10; i++ )  
{  
    fseek(fp, ( 9 - i ) * sizeof( i ), SEEK_SET);  
    fread(&k, sizeof( i ), 1, fp);  
    printf("k=□%d\n", k);  
}  
fclose(fp);  
return 0;  
}
```




```
z123456@turing:~$ myprog dog cat frog
```



Operacje na łańcuchach znaków

wer. 7

Wojciech Myszka

Katedra Mechaniki, Inżynierii Materiałowej i Biomed

2020-05-20 08:40:08 +0200



Łańcuch znaków

1. Z łańcuchów znaków korzystamy powszechnie.
2. Najprostszy przykład:

```
printf("Ala_ma_kota");
```

3. Łańcuchem znaków (ang: *string*) jest każdy napis zawarty w cudzysłowach.
4. Łańcuch znaków jest **tablicą** typu **char**, w której na ostatniej pozycji znajduje się znak null (znak o kodzie zero).

```
char *tekst = "Jakis_tam_tekst";  
printf("%c\n", "przyklad"[0]);  
    /* wypisze p – znaki w napisach sa  
       numerowane od zera */  
printf("%c\n", tekst[2]); /* wypisze k */
```

Łańcuch znaków cd I

1. Język C nie robi jakichś wielkich rozróżnień między znakami a cyframi:

```
#include <stdio.h>
int main(void)
{
    char test [] = "test";
    int i;
    printf("%ld\n", sizeof(test));
    for (i = 0; i < sizeof(test); i++)
        printf("%d\n", test[i]);
    return 0;
}
```

2. Efekt działania tego programu

Łańcuch znaków cd II

5

116

101

115

116

0



Deklaracje

Poniższe deklaracje są (w zasadzie równoważne)

```
char *tekst = "Jakis_tam_tekst";  
/* Umieszcza napis w obszarze danych  
programu i przypisuje adres */  
char tekst[] = "Jakis_tam_tekst";  
/* Umieszcza napis w tablicy */  
char tekst[] = {'J', 'a', 'k', 'i', 's', \  
'_','t','a','m','_','t','e','k', \  
's','t','\0'};  
/* Tekst to taka tablica jak kazda inna */
```

Jednak w pierwszym przypadku tekst jest odsyłaczem do **stałej!** Elementów tablicy nie można zmieniać!

Operacje na łańcuchach

Plik nagłówkowy `string.h` zawiera definicje stałych i funkcji związanych z operacjami na łańcuchach tekstów.

```
void *memcpy(void *, const void *, int, \
size_t);
void *memchr(const void *, int, size_t);
int memcmp(const void *, const void *, \
size_t);
void *memcpy(void *, const void *, \
size_t);
void *memmove(void *, const void *, \
size_t);
void *memset(void *, int, size_t);
char *strcat(char *, const char *);
char * strchr(const char *, int);
int strcmp(const char *, const char *);
int strcoll(const char *, const char *);
char *strcpy(char *, const char *);
size_t strcspn(const char *, const char *);
char *strdup(const char *);
```

```
char *strerror(int);
size_t strlen(const char *);
char *strncat(char *, const char *, \
size_t);
int strncmp(const char *, const char *, \
size_t);
char *strncpy(char *, const char *, \
size_t);
char *strpbrk(const char *, const char *);
char *strrchr(const char *, int);
size_t strspn(const char *, const char *);
char *strstr(const char *, const char *);
char *strtok(char *, const char *);
char *strtok_r(char *, const char *, \
char **);
size_t strxfrm(char *, const char *, \
size_t);
```

Porównywanie ciągów znaków

1. Porównywanie pojedynczych znaków — na podstawie ich kodów ASCII
2. Dla ciągów znaków "aaa" > "aa" > "a" > ""; "ab" > "aa"; "a" > "A"
3. Funkcja *strcmp* służy do porównywania dwu ciągów znaków; ma dwa parametry i zwraca -1 gdy pierwszy ciąg znaków jest mniejszy od drugiego, 0 gdy są równe i $+1$, gdy pierwszy ciąg jest większy od drugiego.
4. Funkcja *strncmp* (o trzech parametrach, trzeci parametr wskazuje ile znaków ma być porównanych) może być użyta, gdy nie chcemy porównywać całych ciągów znaków.

Kopiowanie znaków

1. Do kopiowania służy funkcja *strcpy* o dwu parametrach; ciąg znaków zawarty w drugim parametrze kopiowany jest do pierwszego parametru. Do obowiązku programisty należy zapewnienie aby tablica będąca pierwszym parametrem miała wystarczającą ilość miejsca na pomieszczenie całego zapisu!
2. Drugi wariant funkcji *strncpy*; dodatkowy, trzeci parametr mówi ile znaków ma być skopiowanych — (ale zawsze trzeba pamiętać o miejscu na znak null znajdującym się na końcu napisu). Znaki z drugiego parametru kopiowane są do końca łańcucha, chyba że tekst jest dłuższy niż liczba znaków do skopiowania; w tym przypadku programista musi dodać znak NULL „ręcznie” na końcu skopiowanego tekstu.

Łączenie napisów I

1. Do łączenia napisów służy funkcja *strcat* (nazwa pochodzi od STRing conCATenate).
2. Podobnie jak w poprzednich przypadkach jest również drugi wariant *strncat*
3. W wersji standardowej Zawartość drugiego parametru funkcji dopisywana jest na końcu zawartości pierwszego.

```
char tekst[80] = "";  
strcat(tekst, "ala");  
strcat(tekst, "_ma_");  
strcat(tekst, "kota");  
puts(tekst);
```

4. Pierwszy parametr funkcji musi mieć dostateczną ilość miejsca na pomieszczenie połączonych ciągów znaków!
5. W drugim wariantcie występuje trzeci parametr mówiący ile znaków ma być dołączonych.

Długość ciągu znaków

1. Funkcja *strlen* podaje długość ciągu znaków (bez znaku NULL)



Wyszukiwanie

1. Funkcja *strchr* (gdzie pierwszym parametrem jest ciąg znaków a drugim pojedynczy znak) zwraca numer znaku zgodnego z poszukiwanym.
2. Funkcja *strrchr* poszukuje od prawej do lewej.
3. Funkcja *strstr* (o dwu parametrach) wyszukuje pierwszego wystąpienia ciągu znaków będącego drugim parametrem w pierwszym.
4. Funkcja *strtok* może służyć do podziału pierwszego parametru na ciąg żetonów; zakładamy, że żetony (tokeny) oddzielone są zadanyim znakiem (drugi parametr).

Konwersje

1. *atol*, *strtol* — zamienia łańcuch na liczbę całkowitą typu **long**
2. *atoi* — zamienia łańcuch na liczbę całkowitą typu **int**
3. *atoll*, *strtoll* — zamienia łańcuch na liczbę całkowitą typu **long long** (64 bity); dodatkowo istnieje przestarzała funkcja *atolq* będąca rozszerzeniem GNU,
4. *atof*, *strtod* — przekształca łańcuch na liczbę typu **double**

Funkcje *ato** nie wykrywają błędów konwersji

Funkcje zdefiniowane są w pliku `stdlib.h`

Różne informacje

Plik nagłówkowy `ctype.h` zawiera definicje funkcji (i różnych stałych) związanych z rozpoznawaniem typów informacji

```
int  isalnum(int);  
int  isalpha(int);  
int  isascii(int);  
int  isblank(int);  
int  iscntrl(int);  
int  isdigit(int);  
int  isgraph(int);  
int  islower(int);  
int  isprint(int);
```

```
int  ispunct(int);  
int  isspace(int);  
int  isupper(int);  
int  isxdigit(int);  
int  toascii(int);  
int  tolower(int);  
int  toupper(int);  
int  _toupper(int);  
int  _tolower(int);
```

```
int atoi( const char * string );  
long atol( const char *s );  
double atof( const char* string );
```

Parametrem funkcji jest ciąg znaków zawierający napis, który ma być skonwertowany

Funkcje z rodziny strtol*

```
long int strtol(const char *str, char **endptr, int base);  
unsigned long int strtoul(const char *str, char **endptr, int base);  
double strtod(const char *str, char **endptr);
```

Pierwszym parametrem funkcji jest napis podlegający konwersji, drugim jest znak ograniczający napis (może to być znak NULL). Trzecim parametrem w przypadku funkcji konwersji do postaci całkowitej jest podstawa zapisu liczb (zawarta pomiędzy 2 a 36). Specjalna wartość 0 pozwala automatycznie rozpoznawać liczby postaci 0nnnnn (gdzie n to cyfra od zera do 7) jako ósemkowe a liczby postaci 0xuuuuu jako szesnastkowe (u to cyfry od zera do 9 oraz a — f lub A — F)

Unicode I

1. Jeżeli ograniczyć się do kodowania jednobajtowego (255 znaków) nie ma właściwie żadnych problemów, poza tym, że program musi być uruchamiany w odpowiednim środowisku. Dostępne kodowania znaków to ISO-8859-n (n od 1 do 9), cp-12xx, i tak dalej, i tak dalej. . .
2. Standardowe funkcje porównywania ciągów znaków nie będą działały!
3. Uniwersalny system kodowania znaków Unicode został dopuszczony do użytku w standardzie C99
4. Z Unicode też nie jest łatwo, mamy kilka rodzajów. Zestaw znaków Unicode obejmuje ponad 900 tys. symboli. Zapisać to można na 3 bajtach.
5. Ze względów technicznych przyjęto 4 bajty jako podstawowa wielkość znaku (bo łatwo jedną instrukcją) wybrać z pamięci.
6. W praktyce to za dużo, stąd warianty
 - 6.1 UTF-8 — od 1 do 6 bajtów (dla znaków poniżej 65536 do 3 bajtów) na znak; wszystkie standardowe znaki ASCII na jednym bajcie.

Unicode II

- 6.2 UTF-16 — 2 lub 4 bajty (UTF-32) na znak.
- 6.3 UCS-2 — 2 bajty na znak przez co znaki z numerami powyżej 65 535 nie są uwzględnione
- 7. Domyślne kodowanie dla C to kodowanie zależne od systemu; linux: **czterobajtowe!**.

Unicode III

Co należy zrobić, by zacząć korzystać z kodowania UCS-2

- ▶ powinniśmy korzystać z typu `wchar_t` (ang. „wide character”), jednak jeśli chcemy udostępniać kod źródłowy programu do kompilacji na innych platformach, powinniśmy ustawić odpowiednie parametry dla kompilatorów, by rozmiar był identyczny niezależnie od platformy.
- ▶ korzystamy z odpowiedników funkcji operujących na typie `char` pracujących na `wchar_t` (z reguły składnia jest identyczna z tą różnicą, że w nazwach funkcji zastępujemy „str” na „wcs” np. `strcpy` — `wcscpy`; `strcmp` — `wcscmp`)
- ▶ jeśli przyzwyczajeni jesteśmy do korzystania z klasy `string`, powinniśmy zamiast niej korzystać z `wstring`, która posiada zbliżoną składnię, ale pracuje na typie `wchar_t`.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char napis1[30] = "Ala_ma_kota";
    char napis2[30] = "Ala_ma_kotę";
    printf( "napis1: %d\n", (int) strlen( napis1 ) );
    printf( "napis2: %d\n", (int) strlen( napis2 ) );
    return 0;
}
```

Wyniki:

napis1: 11

napis2: 12



Polskie literki

```
#include <stdio.h>
#include <string.h>
#include <wchar.h>
int main(void)
{
    char napis1[30] = "Ala_ma_kota";
    char napis2[30] = "Ala_ma_kotę";
    wchar_t napis3[12] = L"Ala_ma_kotę";
    printf( "napis1:_%d\n", (int) strlen(napis1) );
    printf( "napis2:_%d\n", (int) strlen(napis2) );
    printf( "napis3:_%d,_%sizeof_napis3_%d\n",
            (int) wcslen(napis3),
            (int) sizeof( napis3 ) );
    return 0;
}
```

Wyniki

napis1: 11

napis2: 12

napis3: 11, sizeof napis3 48



Polskie literki

```
#include <stdio.h>
#include <string.h>
#include <wchar.h>
int main(void)
{
    int i;
    wchar_t m[2] = L"A";
    union ccc{
        char n[8];
        wchar_t N[2];
    } c;
    c.N[0] = m[0];
    c.N[1] = m[1];
    for (i=0; i<8; i++)
        printf("%d└─┘%x\n", i, c.n[i]);
    return 0;
}
```

0 - 41

1 - 0

2 - 0

3 - 0

4 - 0

5 - 0

6 - 0

7 - 0



Polskie literki

```
#include <stdio.h>
#include <string.h>
#include <wchar.h>
int main(void)
{
    int i;
    wchar_t m[2] = L"A";
    union ccc{
        char n[8];
        wchar_t N[2];
    } c;
    c.N[0] = m[0];
    c.N[1] = m[1];
    for (i=0; i<8; i++)
        printf("%d - %x\n", i, c.n[i]);
    return 0;
}
```

```
0 - 4
1 - 1
2 - 0
3 - 0
4 - 0
5 - 0
6 - 0
7 - 0
```

Locale

1. *Locale* to zestaw definicji określających specyficzny dla różnych języków (i krajów) sposób prezentacji różnych informacji.
2. Z jakichś dziwnych powodów problem nie ma zbyt bogatej „literatury”.
3. Jeżeli ktoś chce się zapoznać z pewną realizacją uniksową — mogę polecić bardzo stary tekst [System Operacyjny HP-UX a sprawa polska](#).

Locale

Unix

Zachowanie systemu po wyborze określonego języka zmienia się. Za pomocą kilku zmiennych środowiska można regulować zakres w jakim podporządkowujemy się specyficznym regułom. I tak:

- `LANG` określa używany język,
- `LC_CTYPE` definiuje charakter (litery, cyfry, znaki przestankowe, znaki drukowalne) poszczególnych znaków,
- `LC_COLLATE` określa sposób sortowania,
- `LC_MONETARY` opisuje oznaczanie jednostek monetarnych (symbol waluty, gdzie jest on umieszczany...),
- `LC_NUMERIC` sposób zapisu liczb, znak oddzielający grupy cyfr, znak oddzielający część całkowitą od ułamkowej,
- `LC_TIME` postać podawania daty i czasu,
- `LC_MESSAGES` język używany w wyświetlanych komunikatach,
- `LC_ALL` wszystkie `LC_*`.

1. Sprawa nie jest prosta!
2. Można wszystko zrobić na wiele sposobów.
3. Bardzo często komunikaty wyprowadzanie przez (różne) programy powtarzają się.
4. Powstaje myśl stworzenia „bazy danych” zawierających różne komunikaty oraz ich tłumaczenia na różne języki.
5. Są też gotowe biblioteki zapewniające obsługę takiej bazy danych (oraz jej tworzenie).



Locale

Przykład

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
#include <locale.h>
int main(void)
{
    /*    printf("%s\n", getenv("LANG")); */
    setlocale(LC_ALL, getenv("LANG"));
    printf("%d□\n", wcscoll(L"żółć", L"żółw"));
    return 0;
}
```

Wyniki

```
$ LANG=C ./stringi
```

```
1
```

```
$ LANG=pl_PL.utf8 ./stringi
```

```
-278
```