Obsługa plików

- Plik (ang. *file*) jest wydzielonym fragmentem pamięci (najczęściej dyskowej) posiadającym nazwę.
- Z punktu widzenia języka C plik jest ciągiem bajtów, z których każdy może zostać oddzielnie odczytany.

Widok tekstowy i widok binarny

Zgodnie ze standardem ANSI dwoma sposobami postrzegania plików przez program są widok ***binarny*** i widok ***tekstowy***.
W widoku ***binarnym*** każdy bajt pliku jest dostępny dla programu.
W widoku ***tekstowym*** to, co „widzi" program, może różnić się od tego, co faktycznie znajduje się w pliku. Przyczyną tego jest różnica między reprezentacją np. końca wiersza, stosowaną przez lokalne środowisko a reprezentacją przyjętą w języku C.

Pliki standardowe

Programy w języku C otwierają automatycznie trzy pliki.
Pliki te noszą nazwy: ***standardowe wejście***, ***standardowe wyjście*** oraz ***standardowe wyjście dla błędów***.
Standardowe wejście jest domyślnie podstawowym urządzeniem wprowadzania danych, czyli klawiaturą.
Standardowe wyjście i ***standardowe wyjście dla błędów*** to zwykle ekran monitora.

Standardowe wejście dostarcza do programu dane wejściowe. Jest ono plikiem odczytywanym domyślnie przez getchar (), gets () i scanf ().
Standardowe wyjście jest miejscem, do którego wysyłane są zwykłe dane wyjściowe. Wykorzystują je funkcje putchar(), puts() oraz printf ().
Standardowe wejście i ***wyjście*** są ***buforowane***; dane są przesyłane w dużych porcjach (zwykle po 512 bajtów), a nie po jednym bajcie. Pozwala to znacznie zwiększyć szybkość przesyłania danych.

przykład – program odczytujący plik i zliczający liczbę występujących w nim znaków.

```
/* licz.c – obliczenie ilości znaków w pliku */
#include <stdio.h>
#include <stdlib.h> // prototyp funkcji exit()

// argc - ilość argumentów w wierszu poleceń (command line)
// argv - tablica łańcuchów zawierających argumenty uruchomienia programu
// argv[0] - nazwa programu, argv[1] - 1-wszy arg., argv[2] - 2-gi arg., itd.
int main(int argc, char *argv[])
  {
  int ch;       // przechowuje kolejne odczytywane znaki
  FILE *wp;     // deklaracja wskaźnika do pliku
  long licznik = 0;

  if (argc != 2) {printf("Sposób użycia: %s nazwa_pliku\n", argv[0]); exit(1);}

  if ((wp = fopen(argv[1], "r")) == NULL) {printf("Nie można otworzyć %s\n", argv[1]); exit(1);}

  while ((ch = getc(wp)) != EOF) {putc(ch, stdout); licznik++;}

  fclose(wp); // zamkniecie pliku

  printf("Plik %s zawiera %ld znakow\n", argv[1], licznik);
  return 0;
  }
```

Funkcja ***exit()***

Funkcja **exit()** powoduje natychmiastowe zakończenie programu po uprzednim zamknięciu wszystkich otwartych plików.
Argument funkcji **exit ()** jest przekazywany do systemu operacyjnego.
Ogólnie przyjęty zwyczaj każe przekazywać wartość **0** w przypadku prawidłowego zakończenia programu, a wartość niezerową w przypadku zakończenia będącego wynikiem błędu.
Standard wymaga, aby pomyślne zakończenie sygnalizowała wartość **0** lub stała **EXIT_SUCCESS**, a zakończenie niewłaściwe - stałą **EXIT_FAILURE**. Stałe te wraz z prototypem funkcji exit () znajdują się w pliku nagłówkowym **stdlib.h**.

Funkcja ***fopen()***

```
FILE *fopen(const char *filename, const char *mode);
```

Funkcja **fopen()** otwiera plik o *nazwie* podanej w pierwszym argumencie. Drugim jest łańcuch znaków zwierający litery oznaczające *tryb otwarcia pliku*:

Łańcuch	***Znaczenie***
"r"	Otwiera plik tekstowy do odczytu.
"w"	Otwiera plik tekstowy do zapisu, usuwając zawartość pliku, jeśli istnieje, lub tworząc nowy plik, jeśli nie istnieje
"a"	Otwiera plik tekstowy do zapisu, dopisując nowe dane na końcu istniejącego pliku lub tworząc nowy plik, jeśli plik nie istnieje
"r+"	Otwiera plik tekstowy do uaktualnienia, czyli zarówno do odczytywania, jak zapisywania
"w+"	Otwiera plik tekstowy do uaktualnienia (odczytu i zapisu), usuwając zawartość pliku, jeśli istnieje, lub tworząc nowy plik, jeśli nie istnieje
"a+"	Otwiera plik tekstowy do uaktualnienia (odczytu i zapisu), dopisując nowe dane na końcu istniejącego pliku lub tworząc nowy plik, jeśli plik nie istnieje; odczyt może obejmować cały plik, ale zapis może polegać tylko na dodawaniu tekstu
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"	Jak wyżej, ale otwiera plik w trybie binarnym zamiast tekstowego

Wartość zwracana: wskaźnik do pliku `(FILE *)` lub stała `NULL`, gdy pliku nie udało się otworzyć.

Funkcja ***fclose()***

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

Funkcja `fclose()` zamyka plik reprezentowany przez jej argument, w razie potrzeby opróżniając bufor. Jeśli plik nie zostanie zamknięty, a program zakończy działanie, zmiany nie zostaną zapisane.

Funkcja `fclose()` zwraca wartość 0, jeśli operacja się powiodła; w przeciwnym wypadku wartością zwracaną jest stała `EOF`.

```
  if (fclose(wp)) printf("Blad przy zamykaniu pliku %s\n", argv[l]);
```

Zamknięcie pliku może się nie udać np. jeśli na dysku nie ma wolnego miejsca, odłączony zostanie nośnik (np. pendrive) lub wystąpił błąd we/wy.

Funkcje ***getc()*** *i* ***putc()***

```
int getc(FILE *stream);
```

Funkcja **getc()** (lub **fgetc()**) czyta znak z pliku wskazywanego przez jej argument. Jest plikowym odpowiednikiem funkcji getchar().

Funkcja **getc()** zwraca kod pierwszego znaku ze strumienia. W przypadku końca pliku lub błędu funkcja zwraca wartość `EOF`.

```
int putc(int ch, FILE *stream);
```

Funkcja **putc()** zapisuje znak podany jako pierwszy argument do pliku wskazywanego przez jej drugi argument. Jest plikowym odpowiednikiem funkcji putchar().

Funkcja **putc()** zwraca kod pierwszego zapisywanego znaku lub, w przypadku błędu funkcja zwraca wartość `EOF`.

Pliki standardowe

Plik stdio.h definiuje trzy wskaźniki plikowe przyporządkowane trzem standardowym plikom automatycznie otwieranym przez programy w języku C.

Plik standardowy	***Wskaźnik plikowy***	***Zwykle***
Standardowe wejście	stdin	klawiatura
Standardowe wyjście	stdout	ekran
Standardowe wyjście dla błędów	stderr	ekran

Powyższe wskaźniki należą do typu „wskaźnik do FILE", mogą więc służyć jako argumenty standardowych funkcji wejścia/wyjścia np.

```
c = getchar();  ⇔  c = getc(stdin);

putchar(c);  ⇔  putc(c, stdout);
```

```c
/* reduktor.c – zmniejsza rozmiar Twoich plików o dwie trzecie! */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>  // dla strcpy(),  strcat()

int main{int argc,  char *argv[])
  {
  FILE *we, *wy;      // deklaracja 2 wskaźników plikowych
  int ch;
  char nazwa[40];     // „pamięć” dla nazwy pliku wyjściowego
  int licznik = 0;

  if (argc < 2)       // sprawdza obecność argumentu
    {fprintf(stderr, "Sposób użycia: %s nazwa_pliku\n", argv[0]); exit(1);}

  if ((we = fopen(argv[1], "r")) = NULL)     // otwiera plik do czytania
    {fprintf(stderr, "Nie mogłem otworzyć pliku \"%s\".\n", argv[1]); exit(2);}

  strcpy(nazwa, argv[1]);                    // kopiuje nazwę pliku wejściowego do tablicy
  strcat(nazwa, ".red");                     // dodaje do nazwy .red

  if ((wy = fopen(nazwa, "w")) == NULL)      // otwiera plik do zapisu
    {fprintf(stderr,"Nie można utworzyć pliku wyjsciowego.\n"); exit{3);}

  while ((ch = getc(we)) != EOF) {if (licznik++ % 3 == 0) putc(ch, wy);}   // zapisuje co trzeci znak

  if (fclose(we) !=0 t || fclose(wy) != 0) fprintf(stderr, "Blad przy zamykaniu plików.\n");

  return 0;
  }
```

Funkcje *fprintf()* i *fscanf()*

Funkcje fprintf() i fscanf() działają tak samo, jak printf() i scanf() z tą różnicą, że wymagają one podania dodatkowego, pierwszego argumentu: wkaźnika do pliku.

```
int fprintf(FILE *stream, const char *format,...);
```

```
fprintf(stdout, "format" ,...) ⇔ printf("format" ,...)
```

```
int fscanf(FILE *stream, const char *format, ...);
```

```
fscanf(stdin, "format" ,...) ⇔ scanf("format" ,...)
```

Funkcje *sprintf()* i *sscanf()*

Funkcje sprintf() i sscanf() działają tak samo, jak printf() i scanf() z tą różnicą, że wymagają one podania dodatkowego, pierwszego argumentu: wkaźnika do pierwszego elementu tablicy znakowej.

```
int sprintf(char *str, const char *format, ...);
```

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

```c
/* dodajsl.c – korzysta z wprintf(), fscanf() i rewind() */
#include <stdio.h>
#include <stdlib.h>
#define MAX 40

int main(void)
  {
  FILE *wp;
  char słowa[MAX];
  if ((wp = fopen("slowka", "a+")) == NULL) // otwarcie pliku w trybie a+
     {fprintf(stdin, "Nie mogę otworzyć pliku \"slowka\".\n"); exit(1);}
  puts("Podaj słowa, które maja zostać dodane do pliku;");
  puts("Aby zakończyć, wcisnij Enter na początku wiersza.");
  while(gets(słowa) != NULL && słowa[0] != '\0') fprintf(wp, "%s ", słowa);
  puts("Zawartość pliku:");
  rewind(wp);                                  // powrót do początku pliku
  while(fscanf(wp, "%s", słowa) == 1) puts(słowa);  // wypisanie zawartości pliku
  if (fclose(wp) != 0) fprintf(stderr, "Blad przy zamykaniu pliku.\n");
  return 0;
  }
```

Powyższy program zapisuje słowa w pliku.
Dzięki użyciu trybu "a+" program może zarówno odczytywać dane z całego pliku, jak i dopisywać je na jego końcu.
Przy pierwszym uruchomieniu program tworzy plik `slowka` i umożliwia umieszczenie w nim słów.
Przy kolejnych uruchomieniach program dopisuje nowe słowa do poprzedniej zawartości.
Instrukcja `rewind()` powoduje powrót do początku pliku tak, aby ostatnia pętla `while` mogła wyświetlić jego zawartość.
Argumentem funkcji `rewind()` jest wskaźnik do pliku.

Funkcje *fgets()* i *fputs()*

```
char *fgets(char *str, int size, FILE *stream);
```

Funkcja `fgets()` czyta kolejne znaki z pliku wskazywanego przez `stream` i umieszcza je w tablicy znakowej wskazywanej przez `str`. Czytanie przerywa, gdy przeczyta `size-1` znaków, natrafi na koniec pliku lub znak końca linii (w odróżnieniu do `gets()`, znak ten jest zapisywany do `str`). Na końcu `fgets()` dopisuje znak `'\0'`.

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

Funkcja `fputs()` wpisuje łańcuch `s` do pliku wskazywanego przez zmienną `stream`. W przeciwieństwie do funkcji `puts()`, nie dodaje znaku nowej linii `'\n'`.

Dostęp niesekwencyjny: *fseek()* i *ftell()*

```
/* odwroc.c – wyświetla zawartość pliku w odwrotnej kolejności */
#include <stdio.h>
#include <stdlib.h>
#define CNTL_Z '\032'  // znacznik EOF w plikach tekstowych DOS
#define DLAN 50

int main(void)
  {
  char plik[DLAN];
  char ch;
  FILE *wp;
  long licznik, koniec;

  puts("Podaj nazwę pliku:"); gets(plik);
  //                      v   -   tryb binarny tylko do odczytu
  if ((wp = fopen(plik, "rb")) == NULL) {printf("Nie mogę otworzyć %s\n", plik); exit(1);}
  fseek(wp, 0L, SEEK_END);          // przejdź do końca pliku
  koniec = ftell(wp);               //
  for (licznik = 1L; licznik <= koniec; licznik++)
    {
    fseek(wp, -licznik, SEEK_END);  // idz do tylu
    ch = getc(wp);
    if (ch != CNTL_Z && ch != '\r') putchar(ch);
    }
  putchar ('\n');
  fclose (wp);
  return 0;
  }
```

Funkcje *fseek()* i *ftell()*

```
int fseek(FILE *file, long offset, int mode);
```

Funkcja fseek() ustawia pozycję wskaźnika do pliku file na offset bytów względem wartości argumentu mode.
Jeśli mode jest równy SEEK_SET, to offset liczony jest od początku pliku.
Dla mode równego SEEK_CUR , to nastąpi przesunięcie o offset od aktualnej pozycji wskaźnika pliku,
a dla SEEK_END wskaźnik pliku przesuwany jest o offset od końca pliku.

```
long ftell(FILE *file);
```

Funkcja `ftell()` zwraca aktualną pozycję wskaźnika pliku.

Potencjalny problem związany z `fseek()` i `ftell()` polega na tym, iż funkcje te ograniczają rozmiary plików do wartości, które mogą być wyrażone za pomocą typu `long`. Standard ANSI C wprowadził dwie nowe funkcje przystosowane do współpracy z plikami o większych rozmiarach. Zamiast wartości typu `long` przedstawiają one położenie w pliku za pomocą nowego typu pochodnego o nazwie `fpos_t` (ang. *file position type*).

Funkcje *fgetpos()* i *fsetpos()*

```
int fgetpos(FILE *file, fpos_t *pos);
```

Funkcja `fgetpos()` umieszcza w zmiennej wskazywanej przez `pos` aktualną pozycję wskaźnika do pliku `file`. Funkcja zwraca `0` gdy funkcja wykonała się pomyślnie a wartość stałej `EOF` w przypadku wystąpienia błędu.

```
int fsetpos(FILE *file, fpos_t *pos);
```

Funkcja `fsetpos()` zmienia aktualną pozycję wskaźnika do pliku `file` na wartość wskazywaną przez `pos`. Funkcja zwraca `0` gdy funkcja wykonała się pomyślnie a wartość stałej `EOF` w przypadku wystąpienia błędu.

Funkcja *ungetc()*

```
int ungetc(int c, FILE *stream);
```

Funkcja `ungetc()` umieszcza znak określony przez zmienną `c` z powrotem w strumieniu wejściowym. Oznacza to, że znak ten zostanie odczytany jako pierwszy przez najbliższe wywołanie jakiejkolwiek standardowej funkcji wejścia. Wartość zwracana przez funkcję to wycofany znak w razie powodzenia lub `EOF` w razie błędu.

Załóżmy na przykład, że chcesz, aby funkcja pobrała znaki znajdujące się przed następnym znakiem średnika. W tym celu możesz odczytywać znaki do momentu napotkania średnika (za pomocą funkcji `getchar()` lub `getc()`), a następnie „wepchnąć" średnik z powrotem do łańcucha wejściowego przy pomocy funkcji `ungetc()`. Nie można zwrócić w ten sposób znacznika końca pliku (`EOF`). Standard ANSI C gwarantuje możliwość zwrócenia tylko jednego znaku.

Funkcja *fflush()*

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

Funkcja `fflush()` wymusza zapisanie danych znajdujących się w buforach obsługi podanego pliku. Plik pozostaje nadal otwarty. Proces ten nosi nazwę ***opróżnienia bufora*** (ang. *flushing*).
Jeśli wywołanie ma postać: `fflush(NULL);` następuje opróżnienie *wszystkich buforów wyjściowych*.
Funkcja `fflush()` zwraca wartość `0` w przypadku wywołania zakończonego sukcesem lub wartość `EOF`, jeśli wystąpił błąd.

Funkcja int setvbuf(FILE *fp, char *buf, int mode, size_t size)

```
int setvbuf(FILE *fp, char *buf, int mode, size_t size)
```

Funkcja `setvbuf()` tworzy alternatywny bufor przeznaczony do użytku standardowych funkcji wejścia/wyjścia,
Wywołujemy ją po otwarciu pliku i przed wykonaniem na nim jakiejkolwiek innej operacji.
Wskaźnik `fp` określa plik, a `buf` wskazuje na obszar pamięci, który ma zostać użyty. Jeśli wartość argumentu `buf` nie jest równa `NULL`, bufor musisz utworzyć własnoręcznie. Np. przykład, możesz zadeklarować tablicę 1024 wartości typu char i przekazać jej adres do funkcji `setvbuf()`. Jeśli wartością parametru buf jest `NULL`, bufor zostaje utworzony przez funkcję automatycznie. Zmienna `size` określa wielkość bufora. Argument `mode` (tryb) przyjmuje jedną, z następujących wartości:
`_IOFBF` (pełne buforowanie),
`_IOLBF` (buforowanie wierszowe),
`_I0NBF` (brak buforowania).
W przypadku pomyślnego utworzenia bufora funkcja zwraca wartość zero; w przeciwnym wypadku wartość zwracana jest różna od zera.

Binarne wejście/wyjście

Jeśli zapiszemy dane liczbowe do pliku za pocą funkcji `fprintf()`, np.

```
double num = 1.0/3.0;
fprintf(wp, "%.2f", num);
```

zapiszemy zmienną `num` jako czteroznakowy łańcuch: `0.33`. Po zapisaniu zmiennej `num` w ten sposób, nie ma żadnej możliwości odzyskania dalszych miejsc po kropce przy odczytywaniu pliku. Funkcja `fprintf()` przetwarza wartości numeryczne na łańcuchy, co może prowadzić do utraty dokładności.

Najbardziej precyzyjną metodą przechowania liczby jest wyrażenie jej za pomocą tego samego układu bitów, z którego korzysta program. Wartość typu `double` powinna być zapisana w obszarze o rozmiarze typu `double`. Gdy dane w pliku są przedstawione w sposób zgodny z ich reprezentacją w programie, mówimy, że są one zapisane w postaci binarnej.

Funkcja *fwrite()*

```
size_t fwrite(void *ptr,  size_t size,  size_t nmemb, FILE *fp)
```

Funkcja `fwrite()` zapisuje do pliku wskazanego przez `fp`, `nmemb` porcji danych o rozmiarze `size` każda, dane binarne z tablicy wskazanej przez `ptr`.

Następujący kod pozwala zapisać obiekt danych (np. tablicę) o długości 256 bajtów:

```
char bufor[256];
fwrite(bufor, 256, 1, fp);
```

Aby zapisać tablicę składającą się z 10 wartości typu double:

```
double zarobki[10];
fwrite(zarobki, sizeof(double), 10, fp);
```

Funkcja zwraca liczbę pomyślnie zapisanych porcji. Liczba ta powinna być równa `nmemb`, ale może być też mniejsza, jeśli wystąpił błąd zapisu.

Funkcja *fread()*

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *fp)
```

Funkcja fread()ma ten sam zestaw argumentów, co f wri te (). Tym razem jednak ptr jest adresem obszaru pamięci, do którego dane mają zostać wczytane, a f p określa plik wejściowy.

Aby odtworzyć tablicą 10 wartości typu double zapisaną w poprzednim przykładzie:

```
double zarobki[10];
fread(zarobki, sizeof(double), 10, fp);
```

Funkcja zwraca liczbą pomyślnie zapisanych porcji. Liczba ta powinna być równa `nmemb`, ale może być też mniejsza, jeśli wystąpił błąd odczytu lub w przypadku osiągnięcia końca pliku.

Funkcje *feof()* oraz *ferror()*

```
int feof(FILE *fp)

int ferror(FILE *fp)
```

Gdy standardowa funkcja wejścia zwraca wartość `EOF`, z reguły oznacza to, że dotarła ona do końca pliku lub wystąpił błąd odczytu.

Funkcja `feof()` zwraca wartość niezerową, jeśli ostatnie wywołanie funkcji wejścia wykryło koniec pliku; w przeciwnym wypadku wartością zwracaną jest zero.

Funkcja `ferror()` zwraca wartość niezerową, jeśli przy ostatniej operacji wystąpił błąd odczytu lub zapisu.

Łańcuchy znakowe i funkcje łańcuchowe

Stałe łańcuchowe

Stała łańcuchowa (ang. *string constant*) jest ciągiem znaków zawartym pomiędzy znakami cudzysłowu. Ujęte w cudzysłów znaki wraz ze znakiem `\0` dodawanym automatycznie przez kompilator są przechowywane w pamięci jako łańcuch znakowy.

Stałe łańcuchowe należą do klasy *statycznej*. Oznacza to, że stała użyta w funkcji jest przechowywana w pamięci przez cały czas działania programu i tylko w jednej kopii, nawet jeśli funkcja wywoływana jest wiele razy. Całe zdanie w cudzysłowie jest wskaźnikiem do miejsca, w którym zapisany jest łańcuch.

```
/* cudzysl.c – łańcuchy jako wskaźniki */
#include <stdio.h>
int main(void)
  {
  printf("%s, %p, %c\n", "Nie", "jesteśmy", *"kosmicznymi wędrowcami");
  return 0;
  }
```

dane wyjściowe:

```
  Nie,   00410A40,   k
```

Inicjalizacja tablicy znakowej przy pomocy stałej łańcuchowej

```
char tl[] = "Pamietaj, ze musisz sie zmiescic w jednym wierszu.";
```

jest skróconą formą standardowej składni inicjalizacji tablicy:

```
char tl[] = {'P', 'a', 'm', 'i', 'e', 't', 'a', 'j', ',', ' ', 'z', 'e', ' ', 'm', 'u', 's', 'i', 's',
             'z', ' ', 's', 'i', 'e', ' ', 'z', 'm', 'i', 'e', 's', 'c', 'i', 'c', ' ', 'w', ' ',
             'j', 'e', 'd', 'n', 'y', 'm', ' ', 'w', 'i', 'e', 'r', 's', 'z', 'u', '.', '\0', };
```

Deklaracje:

```
char t3[] = "Pamietaj, ze musisz sie zmiescic w jednym wierszu.";

char  *t4 = "Pamietaj, ze musisz sie zmiescic w jednym wierszu.";
```

stwierdzają, że `t3` i `t4` są wskaźnikami do podanych łańcuchow; w obu przypadkach ilość pamięci przeznaczona na łańcuch jest ustalana przez kompilator, ale te formy nie są identyczne.

Główna różnica polega na tym, że nazwa tablicy `t3` jest stałą, a wskaźnik `t4` jest zmienną.

W obu przypadkach można korzystać z notacji tablicowej:

```
for(i = 0; i < 8; i++) putchar(t3[i]); putchar{'\n') ;
for(i = 0; i < 8; i++) putchar(t4[i]); putchar('\n') ;
```

W obu przypadkach możliwe jest dodawanie do wskaźników:

```
for(i = 0; i < 8; i++) putchar(*(t3 + i)); putchar('\n');
for(i = 0; i < 8; i++) putchar(*(t4 + i)); putchar('\n') ;
```

Ale tylko forma wskaźnikowa dopuszcza stosowanie operatora inkrementacji:

```
while (*(głowa) != '\0')  // zatrzymaj na końcu łańcucha
    putchar(*(glowa++));  // wyświetl znak, zwiększ wskaźnik
```

oraz

```
        t4 = t3; // poprawne
        t3 = t4; // niedozwolone!
```

W obu przypadkach możliwe jest:

```
t3[7] = 'M';  // lub *(t3 + 7) = 'M';
t4[7] = 'M';  // lub *(t4 + 7) = 'M';
```

Elementy tablicy `t3` są zmiennymi (chyba że deklaracja tablicy zawiera słowo `const`), ale nie jest zmienną jej *nazwa*.

Jawne określenie rozmiaru tablicy

Odpowiednią ilość miejsca dla łańcucha można również zarezerwować ręcznie.

```
char tl[64] = "Pamietaj, ze musisz sie zmiescic w jednym wierszu.";
```

Należy tylko pamiętać, że liczba elementów musi być co najmniej o 1 większa (z powodu znaku `'\0'`) niż długość łańcucha. Tak jak w przypadku innych tablic statycznych lub zewnętrznych wszystkie niewykorzystane elementy otrzymują domyślnie wartość `'\0'`.

Tablice łańcuchów znakowych

Tablica łańcuchów znakowych pozwala korzystać z kilku różnych łańcuchów, wskazując je przy pomocy indeksu.

przykład:

```
char *mojetab[GRAN] = { "Błyskawiczne dodawanie liczb",
                        "Precyzyjne mnożenie",
                        "Gromadzenie danych",
                        "Scisle wykonywanie instrukcji",
                        "Rozumienie jeżyka C"};
```

Wczytywanie łańcuchów - tworzenie miejsca

Aby wczytać łańcuch do programu, należy przydzielić mu miejsce w pamięci, a następnie pobrać go za pomocą funkcji wejścia. , Ilość miejsca musi być wystarczająca, aby przechować cały ciąg znaków. Np., do kodu:

```
char *imie;
scanf("%s", imię);
```

kompilator prawdopodobnie nie zgłosi zastrzeżeń, ale w wyniku powyższych instrukcji pobrane imię może zostać zapisane w miejscu przechowywania danych lub kodu programu (czyli GDZIEKOLWIEK). Większość programistów uważa tę sytuację za bardzo zabawną, pod warunkiem, że ma ona miejsce nie w ich programach.

funkcja *gets()*

```
char *gets(char *str);
```

Funkcja `gets()` czyta linię ze standardowego wejścia (usuwa ją stamtąd) i umieszcza ją w tablicy znakowej wskazywanej przez `str`.
Ostatni znak linii (znak nowego wiersza - `'\n'`) zastępuje zerem (znakiem `'\0'`).
Wartością funkcji jest `str` w przypadku sukcesu lub `NULL` w przypadku błędu lub natrafienia na koniec pliku. Funkcja nie sprawdza, czy jest miejsce do zapisu w tablicy `str`.

funkcja *puts()*

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

Funkcja wypisuje na standardowe wyjście zawartość tablicy wskazywanej przez `s` do momentu napotkania (jeśli w ogóle) znaku `'\0'`, a następnie znak nowej linii `'\n'`.

```
/* wysw_lan.c – korzystanie z funkcji puts() */
#include <stdio.h>
#define DEF "Jestem z#definiowanym łańcuchem."
int main(void)
  {
  char lan1[80] = "Przypisano mnie tablicy.";
  char *lan2 = "Przypisano mnie wskaźnikowi.";
  puts("Jestem argumentem funkcji puts().");
  puts(DEF);
  puts(lan1);
  puts(lan2);
  puts(&lan1[4]);
  puts(lan2+4);
  return 0;
  }
```

```
dane wyjściowe:
Jestem argumentem funkcji puts().
Jestem z#definiowanym łańcuchem.
Przypisano mnie tablicy.
Przypisano mnie wskaźnikowi.
pisano mnie tablicy.
pisano mnie wskaźnikowi.
```

Funkcje łańcuchowe (*string.h*)

► `char *strcpy(char *s1, const char *s2);`

kopiuje łańcuch (wraz ze znakiem `'\0'`) wskazywany przez `s2` w miejsce wskazywane przez `s1`. Wartością zwracaną jest `s1`.

► `char *strncpy(char *s1, const char *s2, size_t n);`

kopiuje nie więcej niż `n` znaków z łańcucha wskazywanego przez `s2` w miejsce wskazywane przez `s1`. Wartością zwracaną jest `s1`. Nie są kopiowane znaki następujące po znaku `'\0'`, a jeśli łańcuch źródłowy jest krótszy niż `n` znaków, końcówka łańcucha docelowego jest wypełniana znakami `'\0'`. Jeśli łańcuch źródłowy zawiera `n` lub więcej znaków, znak `'\0'` nie jest kopiowany.

► `char *strcat(char *s1, const char *s2);`

Łańcuch wskazywany przez `s2` jest dołączany (wraz ze znakiem `'\0'`) na końcu łańcucha wskazywanego przez `s1`. Pierwszy znak łańcucha `s2` jest umieszczany w miejscu znaku zerowego łańcucha `s1`. Wartością zwracaną jest `s1`.

► `char *strncat(char *s1, const char *s2, size_t n);`

Nie więcej niż pierwsze `n` znaków łańcucha `s2` jest dopisywane do łańcucha `s1`. Pierwszy znak łańcucha `s2` jest umieszczany w miejscu znaku zerowego łańcucha `s1`. Znak zerowy oraz wszelkie następujące po nim znaki nie są kopiowane z łańcucha `s2`. Do łańcucha wynikowego dodawany jest znak zerowy. Wartością zwracaną jest `s1`.

► `int strcmp(const char *s1, const char *s2);`

Funkcja zwraca wartość dodatnią, jeśli łańcuch `s1` znajduje się po łańcuchu `s2` według porządku sortowania komputera, wartość `0`, jeśli łańcuchy są identyczne, oraz wartość ujemną, jeśli pierwszy łańcuch znajduje się przed łańcuchem drugim.

► `int strncmp(const char *s1, const char *s2, size_t n);`

Ta funkcja działa tak samo, jak `strcmp()`, z tym, że porównywanie ulega zatrzymaniu po sprawdzeniu `n` znaków (lub w momencie napotkania znaku zerowego).

► `char *strchr(const char *s, int c);`

Ta funkcja zwraca wskaźnik do pierwszego miejsca w łańcuchu `s` przechowującego znak `c`. (Końcowy znak zerowy jest częścią łańcucha, a więc również on może być obiektem poszukiwań.) w przypadku nieznalezienia znaku funkcja zwraca wskaźnik zerowy.

► `char *strpbrk(const char *s1, const char *s2);`

Ta funkcja zwraca wskaźnik do pierwszego miejsca w łańcuchu `s1` przechowującego jakikolwiek znak znajdujący się w łańcuchu `s2` (puli znaków). W przypadku nieznalezienia takiego znaku funkcja zwraca wskaźnik zerowy.

► `char *strrchr(const char *s, int c};`

Ta funkcja zwraca wskaźnik do ostatniego miejsca w łańcuchu `s` przechowującego znak `c`. (Końcowy znak zerowy jest częścią łańcucha, a więc również on może być obiektem poszukiwań.) W przypadku nieznalezienia znaku funkcja zwraca wskaźnik zerowy.

► `char *strstr(const char *s1, const char *s2);`

Ta funkcja zwraca wskaźnik do pierwszego wystąpienia łańcucha s2 w łańcuchu s1. W przypadku nieznalezienia łańcucha funkcja zwraca wskaźnik zerowy.

► `size_t strlen (const char *s);`

Ta funkcja zwraca liczbę znaków, z wyłączeniem znaku zerowego, z jakiej składa się łańcuch `s`.

… plus kilka innych funkcji …

Przykład: Sortowanie łańcuchów

```
/* sort_lan.c – pobiera łańcuchy i je porządkuje */
#include <stdio.h>
#nclude <string.h>
#define ROZMIAR 81   // granica długości łańcucha, włącznie z \0
#define GRAN 20 // maksymalna liczba wierszy
void srtlan(char *łańcuchy[], int num);    // funkcja sortująca łańcuchy
int main(void)
{
char dane[GRAN][ROZMIAR]; //   tablica przechowująca dane wejściowe
char *wsklan[GRAN];       //   tablica zmiennych wskaźnikowych
int licz =0;    //   licznik danych wejściowych
int k;          //   licznik danych wyjściowych
printf ("Podaj maksymalnie %d wierszy, a ja je uporządkuje.\n",GRAŃ);
printf("Aby zakończyć, wcisnij Enter na początku wiersza.\n");
while (licz < GRAN && gets(dane[licz]) != NULL && dane[licz][0] != '\0')
  {
  wsklan[licz] = dane[licz];   // ustaw wskaźniki na łańcuchy
  licz++;
  }
srtlan(wsklan, licz);          // maszyna sortująca łańcuchy
puts("\nOto uporządkowana lista:\n");
for (k = 0; k < licz; k++) puts(wsklan[k]);    // posortowane wskaźniki
return 0;
}
```

```
/* funkcja sortująca wskaźniki do łańcuchów */
void srtlan(char *lancuchy[], int num)
{
char *temp;
int dol, szuk;
for (dol = 0; dol < num - 1; dol++)
for (szuk = dol + 1; szuk < num; szuk++)
if (strcmp(łańcuchy[dol], łańcuchy[szuk]) > 0)
  {
  temp = łańcuchy[dol];
  lancuchy[dol] = łańcuchy[szuk];
  łańcuchy[szuk] = temp;
  }
}
```

test programu

```
Podaj maksymalnie 20 wierszy, a ja je uporządkuje.
Aby zakończyć, wcisnij Enter na początku wiersza.
Gdybym tak był, gdzie chciałbym być,
To byłbym tam, gdzie nie ma mnie;
Lecz tu, gdzie jestem, musze być,
Nie mogę być zaś tam, gdzie chce.

Oto uporządkowana lista:
Gdybym tak był, gdzie chciałbym być,
Lecz tu, gdzie jestem, musze być,
Nie mogę być zaś tam, gdzie chce.
To byłbym tam, gdzie nie ma mnie;
```

Argumenty wiersza poleceń

Wiersz poleceń (ang. *command line*) jest wierszem, w którym należy coś wpisać, aby uruchomić program.

Argumenty wiersza poleceń to dodatkowe pozycje umieszczone w tym samym wierszu.

```
/* echo.c – funkcja main() z argumentami */
#include <stdio.h>
int main(int argc, char *argv[])
  {
  int licznik;
  printf("Wiersz poleceń zawiera %d argumentów:\n", argc - 1);
  for (licznik = 1; licznik < argc; licznik++) printf{"%d: %s\n", licznik, argv[licznik]);
  printf<"\n");
  return 0;
  }
```