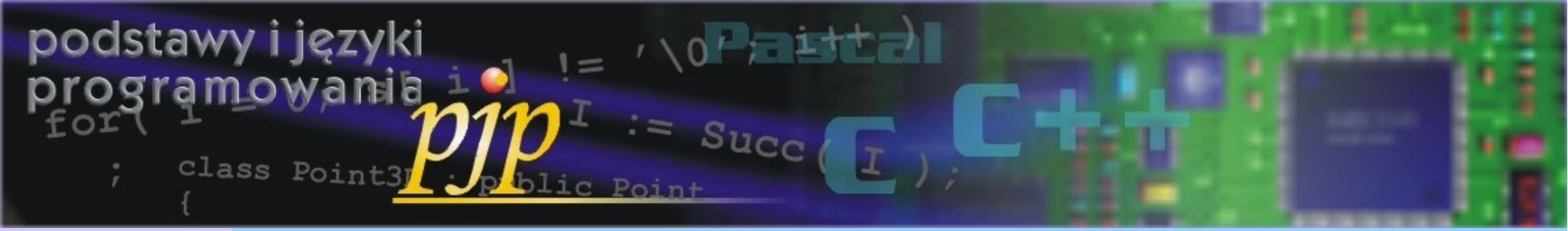


Podstawy programowania w języku C++

Część jedenasta

Reprezentacja i przetwarzanie plików
Konwencja języka C

Autor
Roman Simiński

Kontakt
roman.siminski@us.edu.pl
www.programowanie.siminskionline.pl

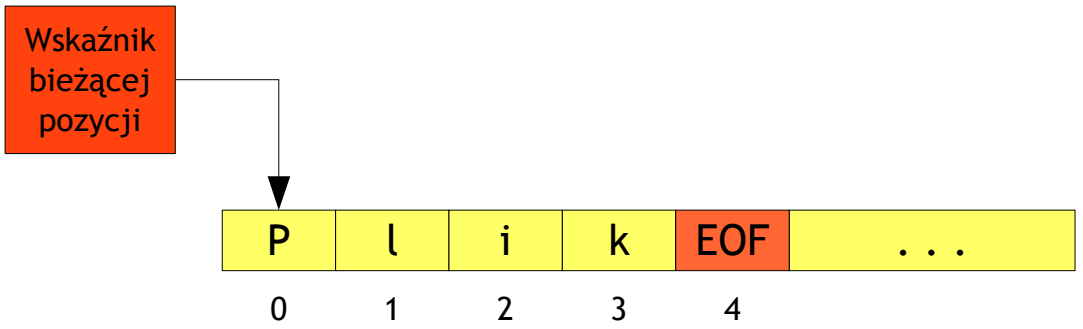
Operacje na plikach w języku C

Język C nie zawiera żadnego wbudowanego typu plikowego! Operacje na plikach nie są częścią języka C.

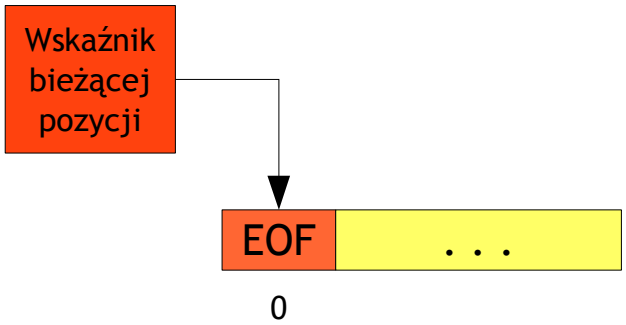
- Przetwarzanie plików realizowane jest zwykle przez funkcje z biblioteki obsługi *standardowego wejścia i wyjścia* (identyfikowanej przez *stdio.h*).
- Można jednak korzystać z funkcji niższego poziomu (np. *io.h*) lub napisać własne funkcje.
- Operacje na standardowym wejściu i wyjściu są *buforowane* — dzieje się to bez dodatkowego żadnego udziału programisty.

Reprezentacja plików w postaci strumieni

- Plik jest reprezentowany przez strumień znaków (bajtów) o zmiennej długości. Koniec strumienia identyfikowany jest znacznikiem końca pliku — EOF.
- Z każdym strumieniem związany jest wskaźnik bieżącej pozycji — od tej pozycji realizowane będzie czytanie lub pisanie.
- Każdy zapis i odczyt zmienia wskaźnik bieżącej pozycji.
- Z każdym strumieniem związany jest znacznik osiągnięcia końca pliku oraz znacznik błędu.



Plik pięcioelementowy



Plik pusty

Otwarcie pliku — tryby otwarcia

Strumienie mogą być otwierane w trybie:

- *Binarnym* — strumień jest ciągiem jednakowo traktowanych bajtów, każdy zapis i odczyt realizowany jest bez żadnych konwersji.
- *Tekstowym* — strumień jest ciągiem linii tekstu zakończonych znacznikiem końca linii — znak '\n'. W trakcie odczytu i zapisu do takiego strumienia mogą zachodzić konwersje spowodowane np. różną fizyczną reprezentacją znacznika końca wiersza (np. para \r\n w plikach tekstowych DOS/Windows, pojedynczy znak \n w systemach Unix'owych, \r na komputerach Macintosh).
- Uwaga — w systemach Unix'owych tryb *binarny* i *tekstowy* są *równoważne*.

Otwarcie pliku — definiowanie wskaźnika plikowego

- Aby rozpocząć operacje na plikach należy zadeklarować w programie zmienną stanowiącą „dojście” do takiego pliku.
- W przypadku obsługi standardowych strumieni deklaruje się zmienną wskaźnikową.
- Typem wskazywanym jest FILE, jest to zdefiniowany w pliku nagłówkowym *stdio.h* typ rekordowy, zawierający informacje o otwartym dojściu do pliku.

```
#include <stdio.h>    // Kompilacja w trybie C++ #include <cstdio>
. . .
FILE * fp = NULL;
```

Otwarcie pliku — wykorzystanie funkcji fopen

- Wykorzystanie pliku rozpoczyna operacja jego otwarcia, realizowana zwykle przez funkcję *fopen*. Otwarcie pliku *dane.txt* do odczytu w *trybie tekstowym* może wyglądać następująco:

```
#include <stdio.h>
. . .
FILE * fp = NULL;
. . .
fp = fopen( "dane.txt", "rt" );
if( fp != NULL )
  // Otwarcie OK, wykonaj operacje na pliku
else
  // Otwarcie nieudane, obsługa sytuacji błędnej
```

Lub krócej:

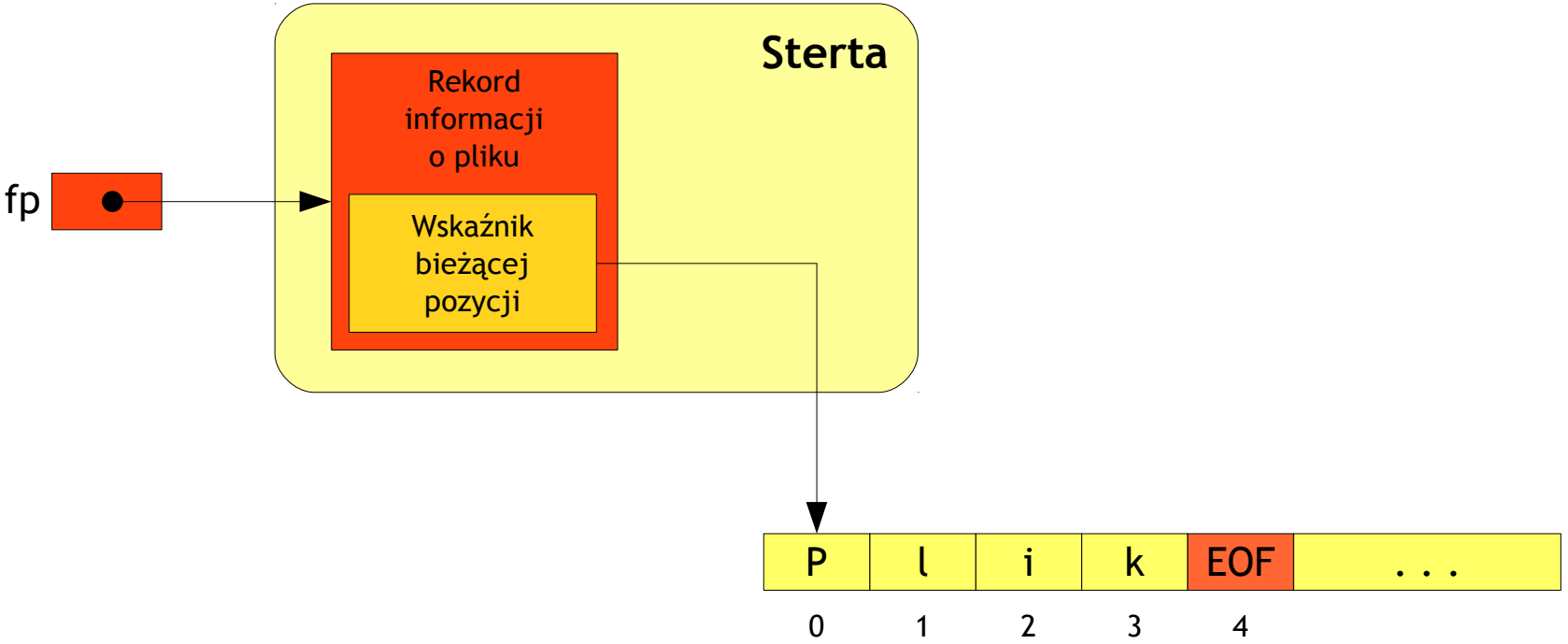
```
#include <stdio.h>
. . .
FILE * fp = NULL;
. . .
if( ( fp = fopen( "dane.txt", "rt" ) ) != NULL )
  // Otwarcie OK, wykonaj operacje na pliku
else
  // Otwarcie nieudane, obsługa sytuacji błędnej
```

Otwarcie pliku — opis funkcji fopen

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

- Otwiera strumień związany z plikiem o nazwie przekazanej parametrem *filename*. Nazwa może zawierać ścieżkę dostępu do pliku. Strumień otwierany jest w trybie *mode*.
- Jeżeli otwarcie zakończyło się sukcesem, funkcja udostępnia wskaźnik do dynamicznie alokowanej struktury typu *FILE*, stanowiącej programową reprezentacje fizycznego pliku.
- Jeżeli pliku nie udało się otworzyć, rezultatem funkcji jest *NULL*.

Otwarcie pliku — działanie funkcji fopen




```
#include <stdio.h>
. . .
FILE * fp = NULL;
. . .
if( ( fp = fopen( "dane.txt", "rt" ) ) != NULL )
  // Otwarcie OK, wykonaj operacje na pliku
else
  // Otwarcie nieudane, obsługa sytuacji błędnej
```

Tryby otwarcia pliku

Specyfikacja typu otwieranego pliku:

t	otwarcie w trybie tekstowym	b	otwarcie w trybie binarnym

Tryb otwarcia:

r	otwarcie istniejącego pliku wyłącznie do odczytu	r+	otwarcie istniejącego pliku do odczytu i zapisu
a	zapis do istniejącego lub utworzenie nowego pliku Ustawienie w pozycji końcowej	a+	odczyt i zapis do istniejącego lub utwo-rzenie nowego pliku Ustawienie w pozycji końcowej
w	utworzenie pliku wyłącznie do zapisu, jeżeli plik istnieje jest obcinany do pliku pustego	w+	utworzenie pliku do odczytu i zapisu, jeżeli plik istnieje jest obcinany do pliku pustego

- Znak + w trybie otwarcia oznacza aktualizację — możliwość czytania i pisania do otwartego strumienia.
- Jednak zapis odczyt i zapis (albo zapis i odczyt) nie mogą po sobie następować bezpośrednio. Należy użyć funkcji „wymiatania” bufora *fflush* lub jednej z funkcji pozycjonowania pozycji — *fseek*, *fsetpos*, *rewind*.

Domyślny typ pliku — binarny czy tekstowy?

- Jeżeli informacja o trybie otwarcia (*t* lub *b*) nie występuje, przyjmowany jest tryb otwarcia zgodnie z wartością globalnej zmiennej *_fmode*.
- Jeżeli *_fmode* posiada wartość *O_BINARY*, plik jest otwierany w trybie binarnym.
- Jeżeli *_fmode* posiada wartość *O_TEXT*, plik jest otwierany w trybie tekstowym.
- Domyślna wartość *_fmode* to *O_TEXT*.
- Symbole *O_TEXT* i *O_BINARY* są zdefiniowane w pliku *fcntl.h*.

Przykłady różnych typów i trybów otwarcia plików

Otwarcie pliku *dane.txt* jako pliku *tekstowego*, wyłącznie do *odczytu*.

```
fp = fopen( "dane.txt", "rt" );
```

Otwarcie pliku *tlo.bmp* jako pliku *binarnego*, wyłącznie do *odczytu*.

```
fp = fopen( "tlo.bmp", "rb" );
```

Otwarcie pliku *podanie.doc* jako pliku *tekstowego*, wyłącznie do *zapisu*, plik ustawiany jest w pozycji końcowej, jeżeli nie istnieje, tworzony jest *nowy*, *pusty*.

```
fp = fopen( "podanie.doc", "at" );
```

Otwarcie pliku *image.jpg* jako pliku *binarnego*, do *zapisu* i *odczytu*, jeżeli plik istnieje, obcinany jest do pliku *pustego*.

```
fp = fopen( "image.jpg", "w+b" );
```

Zamykanie otwartych plików — fclose

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

- Funkcja zamyka strumień *stream* i zapisuje wszystkie bufory.
- Rezultat *EOF* oznacza błąd zamykania, rezultat równy zero oznacza bezbłędne zamknięcie.
- Pamięć przydzielona strukturze wskazywanej przez wskaźnik *stream* jest zwalniana.

Typowy scenariusz otwarcia i zamknięcia pliku:

```
#include <stdio.h>
. . .
FILE * fp = NULL;
. . .
if( ( fp = fopen( "dane.txt", "rt" ) ) != NULL )
{
  // Otwarcie OK, wykonaj operacje na pliku
  fclose( fp );
}
```

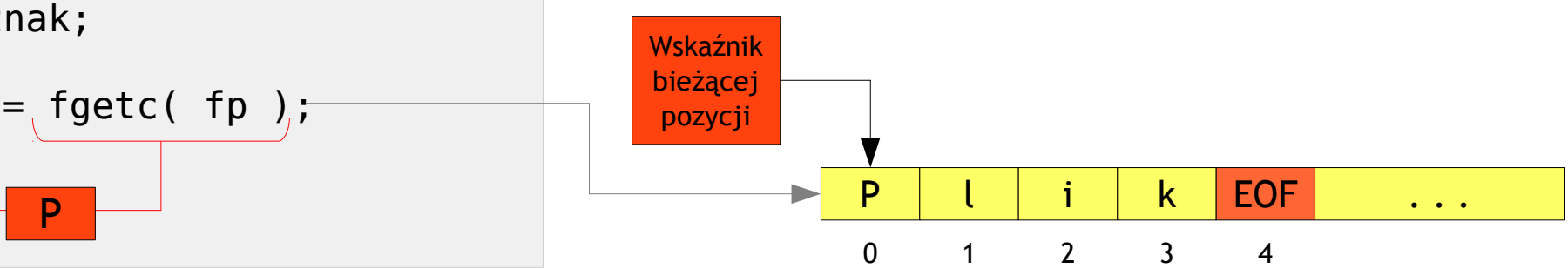
Odczyt pojedynczych znaków

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

- Funkcja pobiera następny znak ze strumienia identyfikowanego przez *stream* i *uaktualnia wskaźnik bieżącej pozycji* w pliku. Znak pobierany jest jako *unsigned char* i przekształcany jest do typu *int*.
- W przypadku napotkania końca strumienia, rezultatem jest wartość *EOF* oraz ustawiany jest *znacznik napotkania końca strumienia*.
- W przypadku wystąpienia błędu odczytu, rezultatem funkcji jest wartość *EOF* oraz ustawiany jest *znacznik błędu strumienia*.

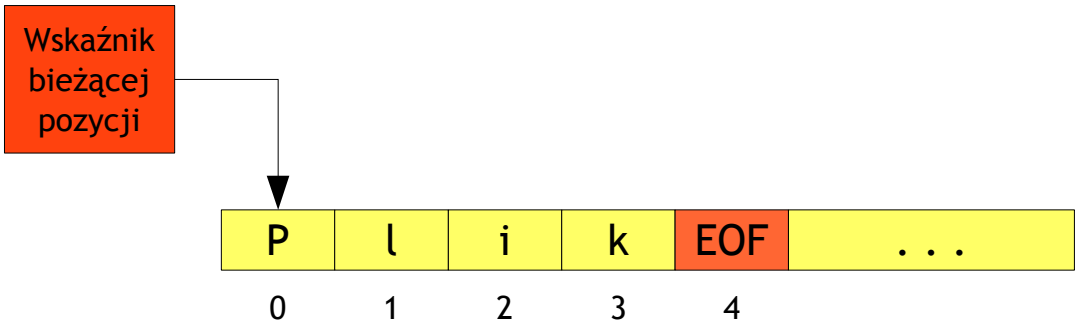
Przykładowe wykorzystanie — odczyt znaku z uprzednio otwartego pliku *fp*:

```
int znak;
. . .
znak = fgetc( fp );
```



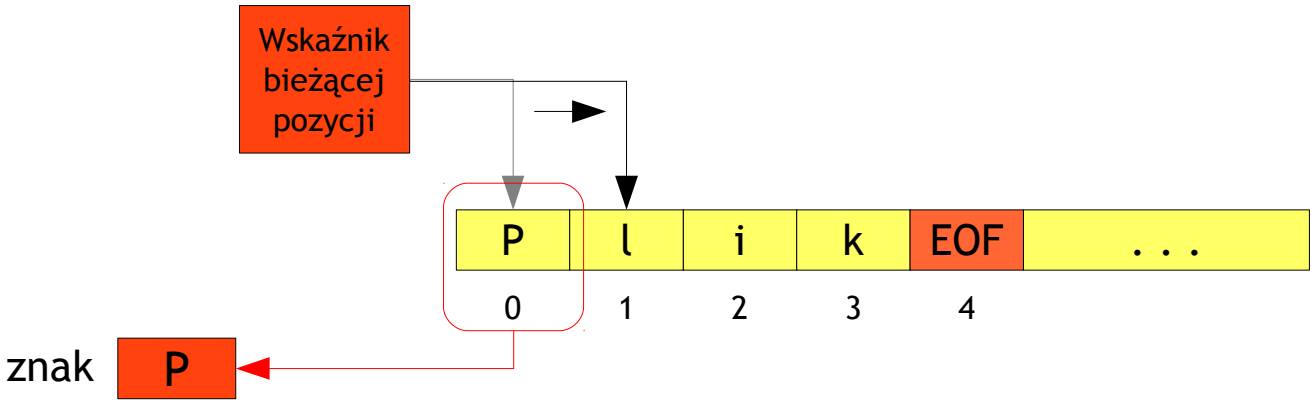
Odczyt pojedynczych znaków, wykorzystanie funkcji fgetc

```
#include <stdio.h>
. . .
FILE * fp = NULL;
. . .
if( ( fp = fopen( "dane.txt", "rt" ) ) != NULL )
{
  int znak;

  znak = fgetc( fp );

  printf( "Przeczytano znak %c", znak );

  fclose( fp );
}
```



Odczyt pojedynczych znaków, wykorzystanie funkcji fgetc

```
#include <stdio.h>
. . .
FILE * fp = NULL;
. . .
if( ( fp = fopen( "dane.txt", "rt" ) ) != NULL )
{
  int znak;

  znak = fgetc( fp );

  printf( "Przeczytano znak %c", znak );

  fclose( fp );
}
```



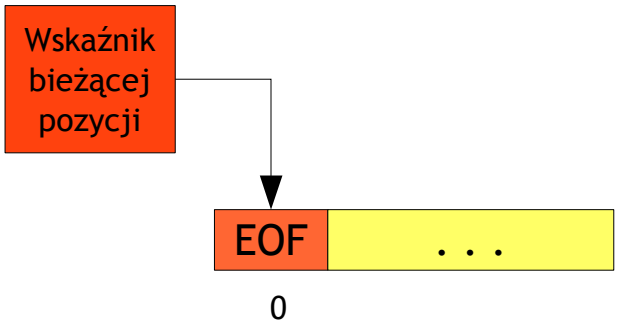
Zapis pojedynczych znaków

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

- Funkcja wyprowadza znak *c* do strumienia *stream* zgodnie ze wskaźnikiem bieżącej pozycji w pliku.
- W przypadku, gdy funkcja *fputc* zakończyła swoje działanie bez błędu, rezultatem funkcji jest znak *c*. W przeciwnym wypadku wartość *EOF*.

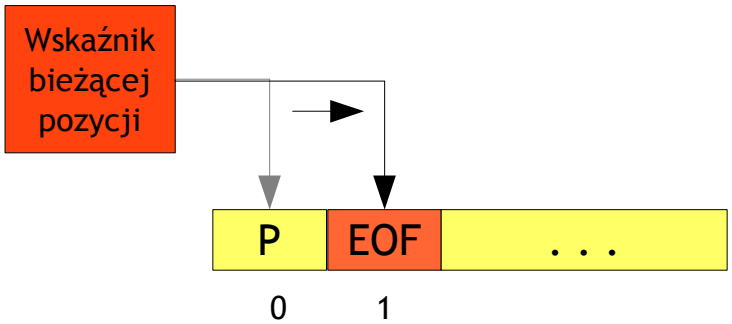
Zapis pojedynczych znaków — wykorzystanie funkcji fputc

```
#include <stdio.h>
. . .
FILE * fp = NULL;
. . .
if( ( fp = fopen( "dane.txt", "wt" ) ) != NULL )
{
  fputc( 'P', fp );

  fclose( fp );
}
```



Zapis pojedynczych znaków — wykorzystanie funkcji fputc

```
#include <stdio.h>
. . .
FILE * fp = NULL;
. . .
if( ( fp = fopen( "dane.txt", "wt" ) ) != NULL )
{
  fputc( 'P', fp );

  fclose( fp );
}
```

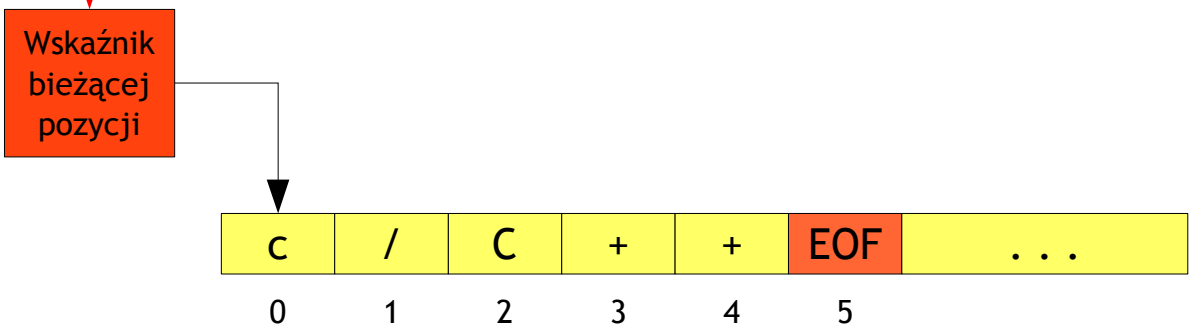


Uwaga — plik powiększa się tylko przy dopisywaniu (zapis na końcu pliku)

```
#include <stdio.h>
. . .
FILE * fp = NULL;
. . .
if( ( fp = fopen( "dane.txt", "r+t" ) ) != NULL )
{
  fputc( 'C', fp );

  fclose( fp );
}
```

Plik istnieje i zawiera napis „c/C++”

W trybie **r** plik jest w pozycji początkowej

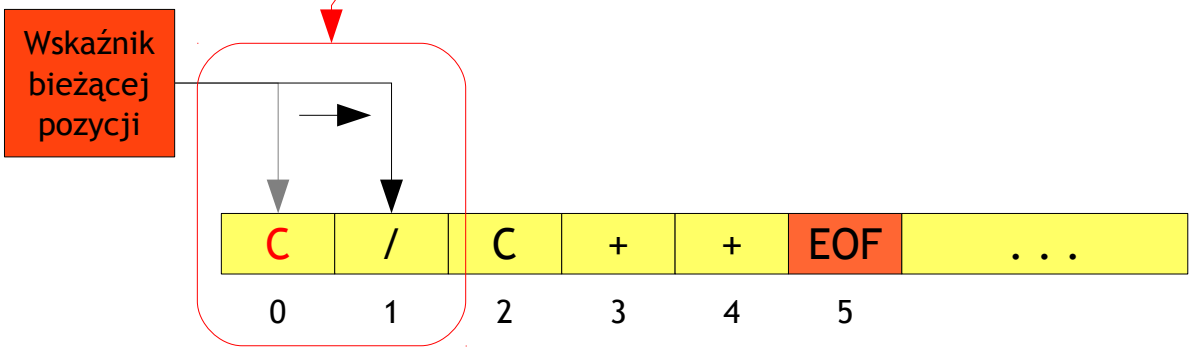


Uwaga — plik powiększa się tylko przy dopisywaniu (zapis na końcu pliku)

```
#include <stdio.h>
. . .
FILE * fp = NULL;
. . .
if( ( fp = fopen( "dane.txt", "r+t" ) ) != NULL )
{
  fputc( 'C', fp );

  fclose( fp );
}
```

Zapis zgodnie z bieżącą pozycją w pliku

Nadpisanie znaku i przesunięcie wskaźnika



Testowanie osiągnięcia znacznika końca pliku

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

- Rezultatem funkcji jest wartość różna od zera, jeżeli strumień jest w pozycji końcowej, zero w przeciwnym wypadku.
- Strumień jest w *pozycji końcowej*, jeżeli w wyniku ostatnio przeprowadzonej operacji *odczytano znacznik końca pliku*.

Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

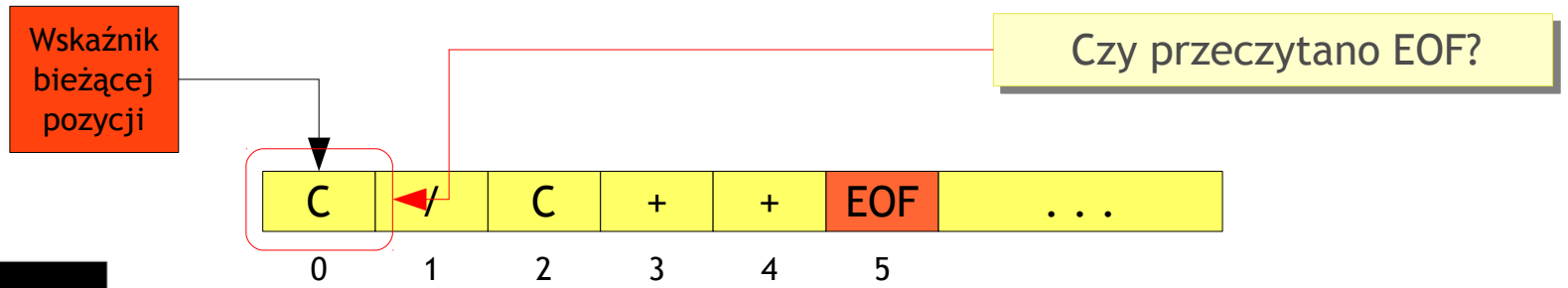
Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```

Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```

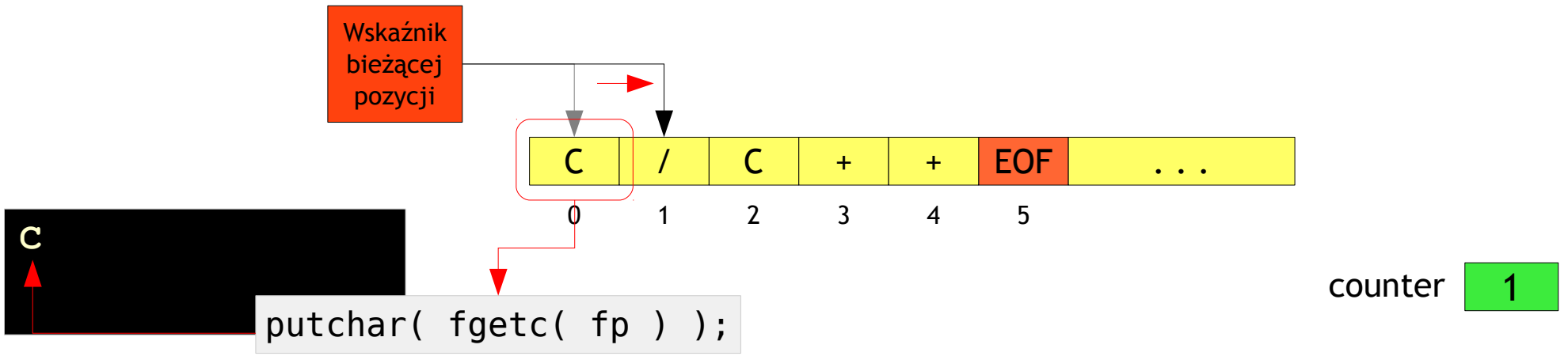


counter 0

Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

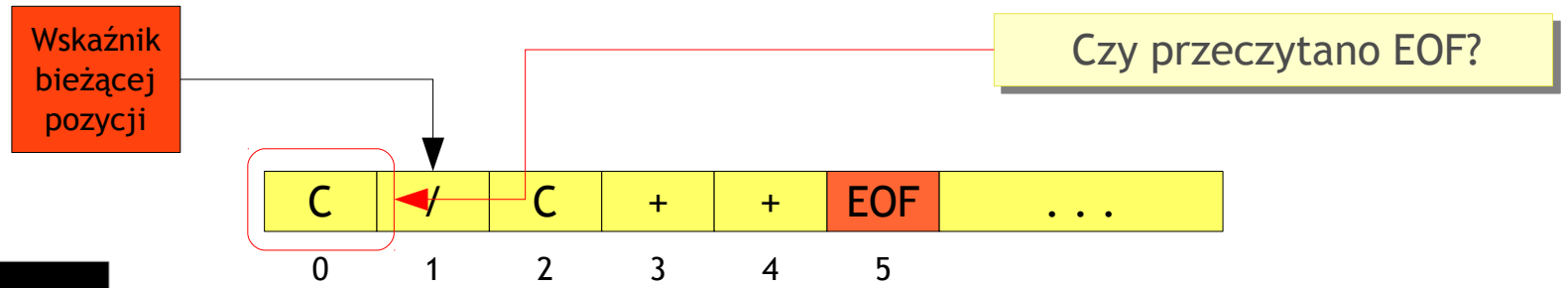
```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```



Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```



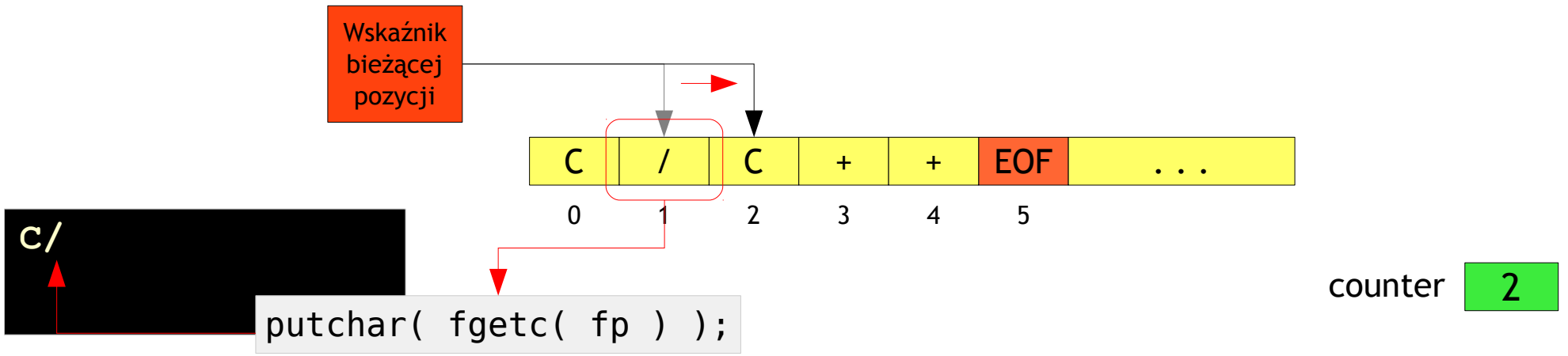

```
C
```

counter 1

Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

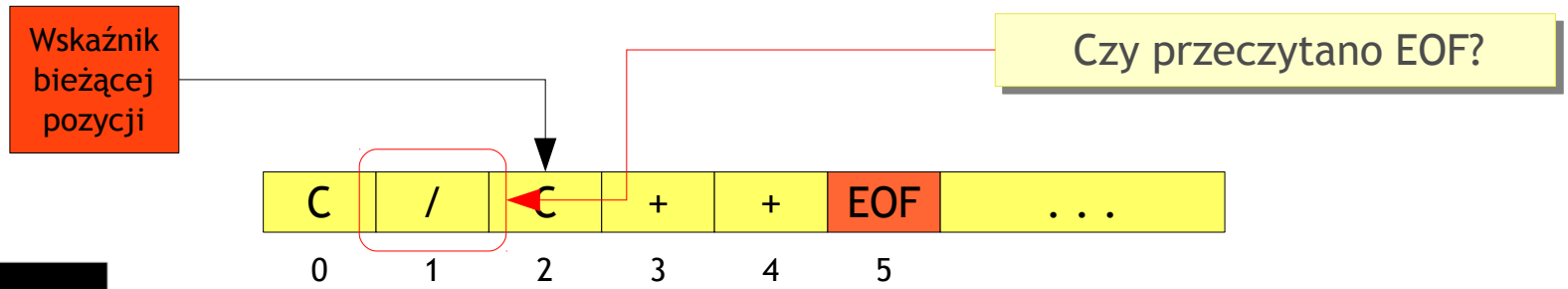
```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```



Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```



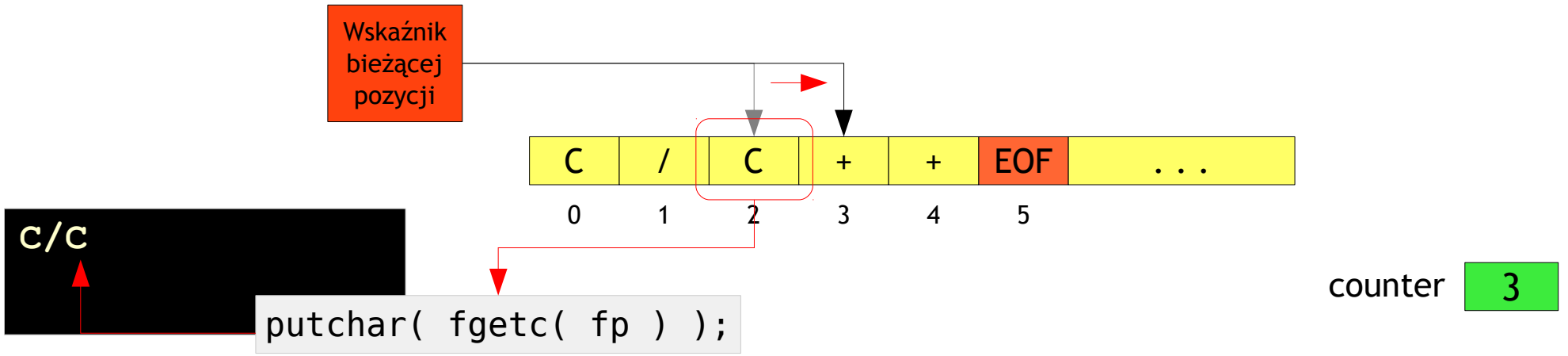

```
c/
```

counter 2

Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

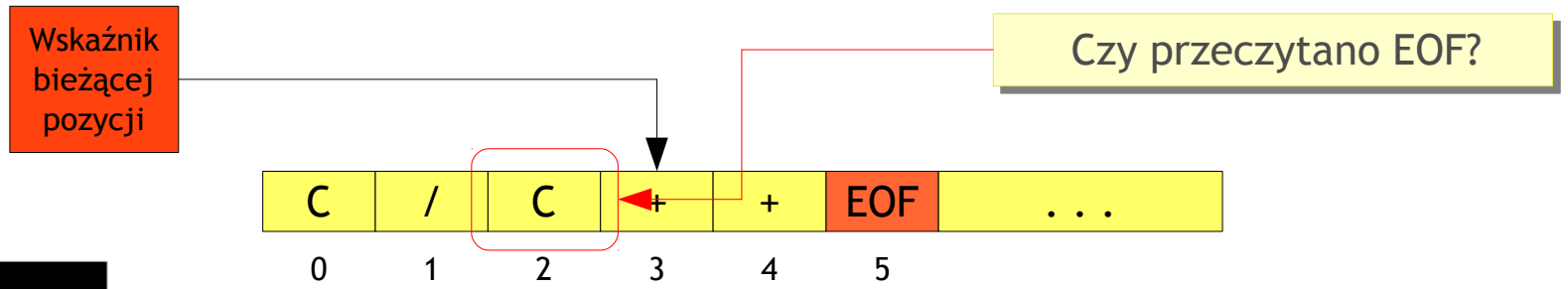
```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```



Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```



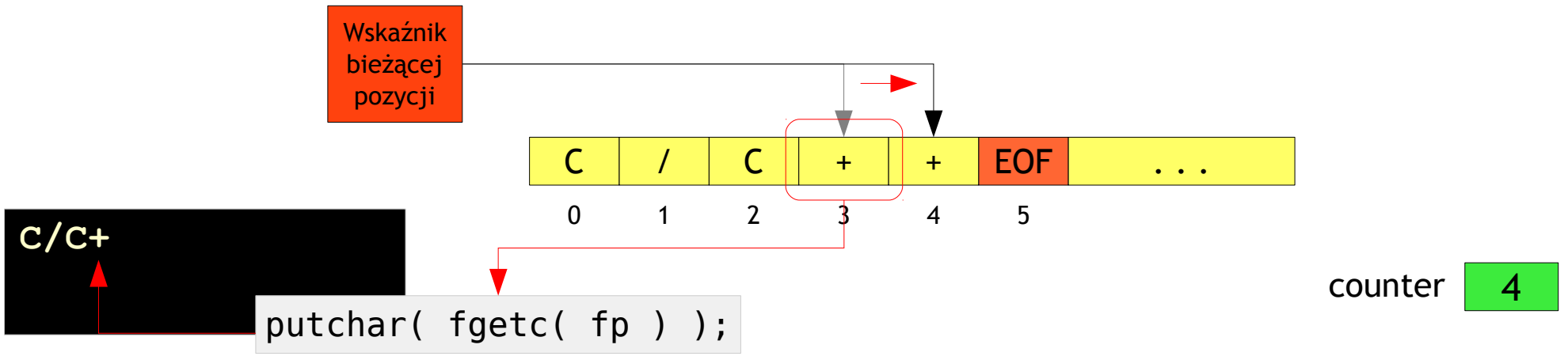

```
C/C
```

counter 2

Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

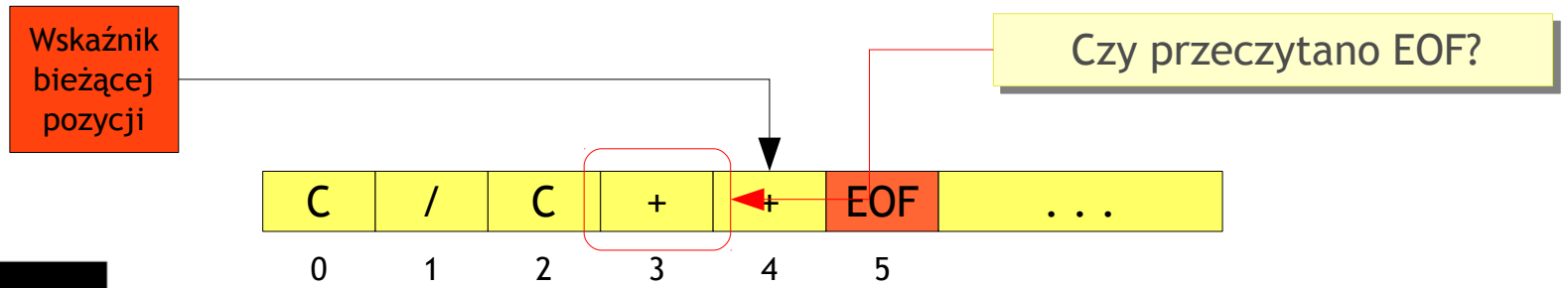
```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```



Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```



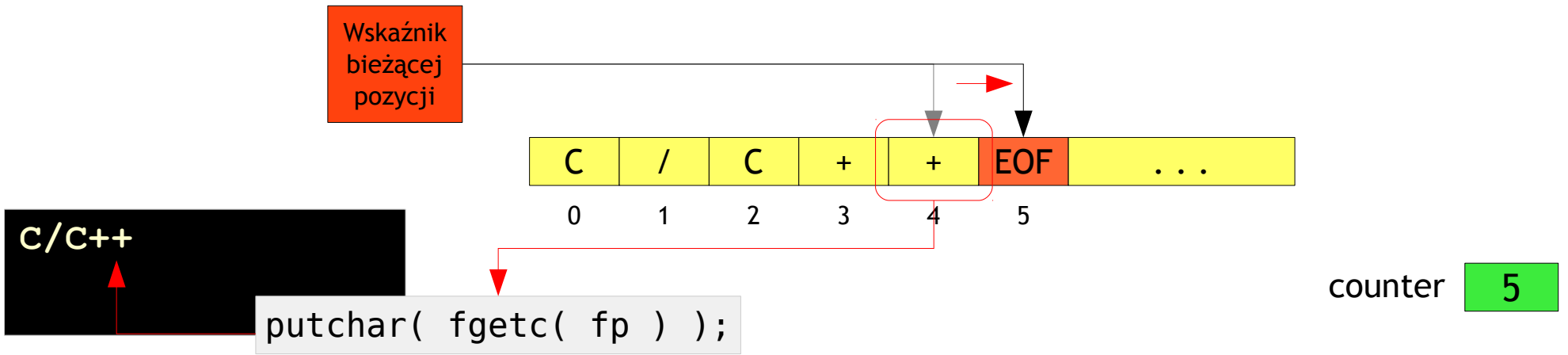

```
C/C+
```

counter 4

Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

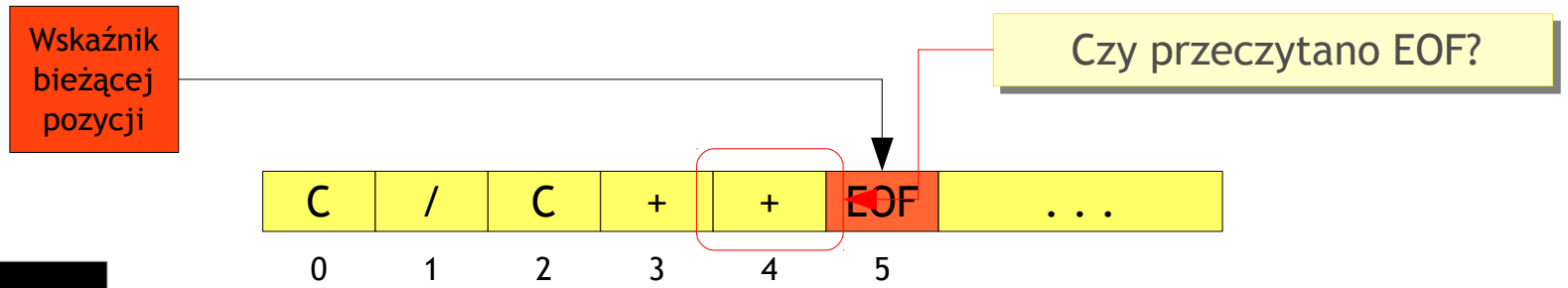
```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```



Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```



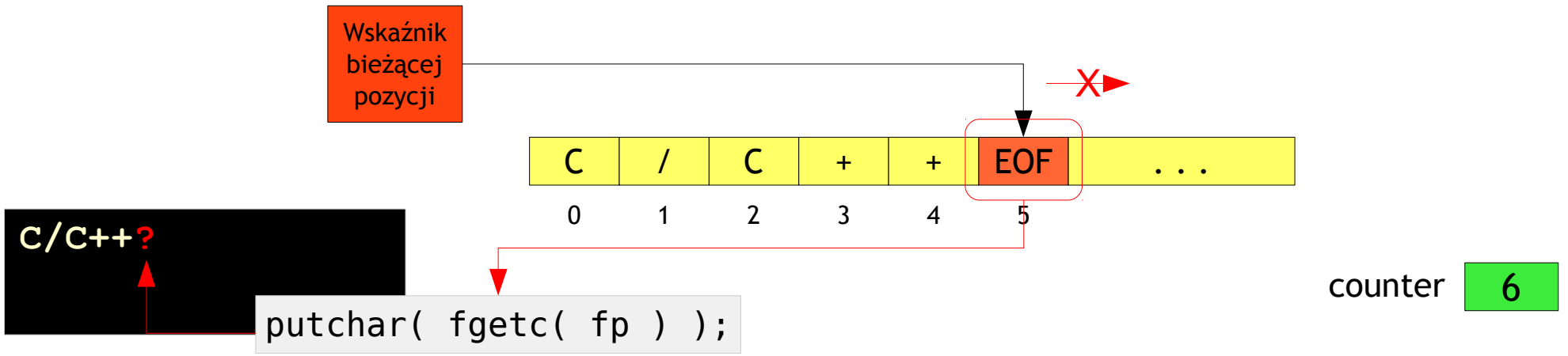
C/C++

counter 5

Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

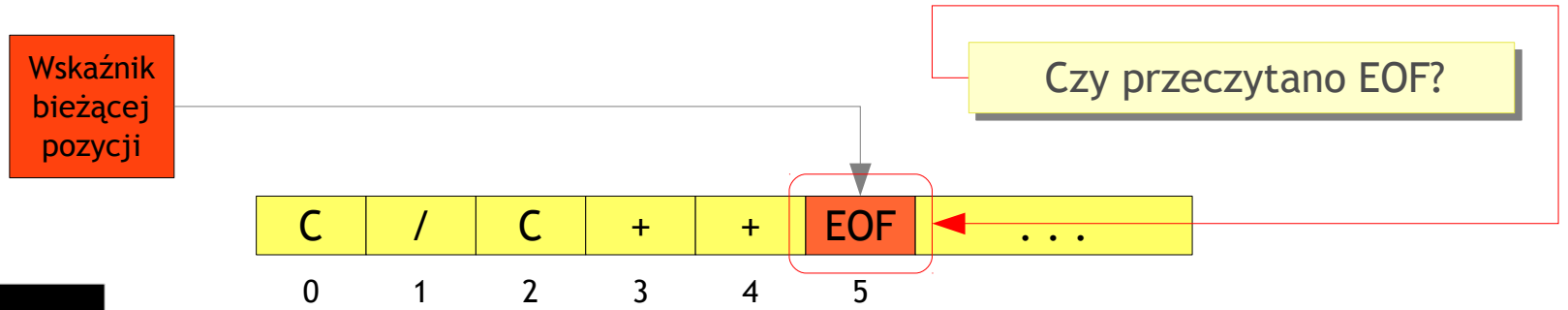
```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```



Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```



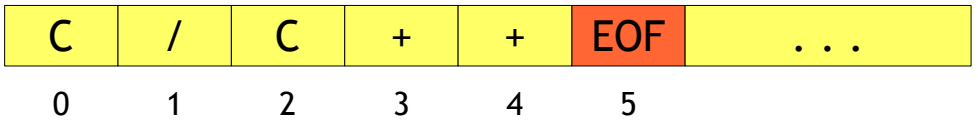
C/C++

counter 6

Sekwencyjne przetwarzanie pliku z wykorzystaniem feof

Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków:

```
FILE * fp;
long int counter = 0;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ! feof( fp ) )
  {
    putchar( fgetc( fp ) );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter - 1 );
}
```




```
C/C++
Liczba znakow w pliku: 5
```

counter 6

Sekwencyjne przetwarzanie pliku bez wykorzystania feof

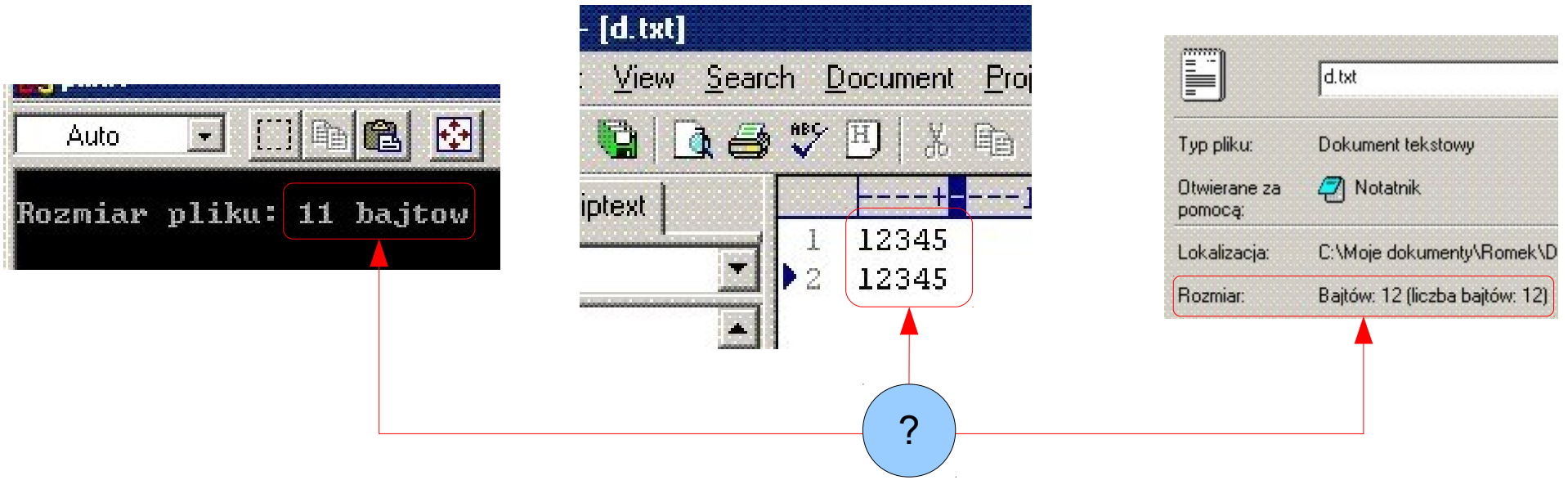
Wypisz do *stdout* zawartość pliku i policz, ile w tym pliku jest znaków, inna wersja:

```
FILE * fp;
long int counter = 0;
int c;

if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  while( ( c = fgetc( fp ) ) != EOF )
  {
    putchar( c );
    counter++;
  }
  fclose( fp );
  printf( "\nLiczba znakow w pliku: %ld", counter );
}
```

Wyznaczanie rozmiaru pliku, Windows i... mały problem

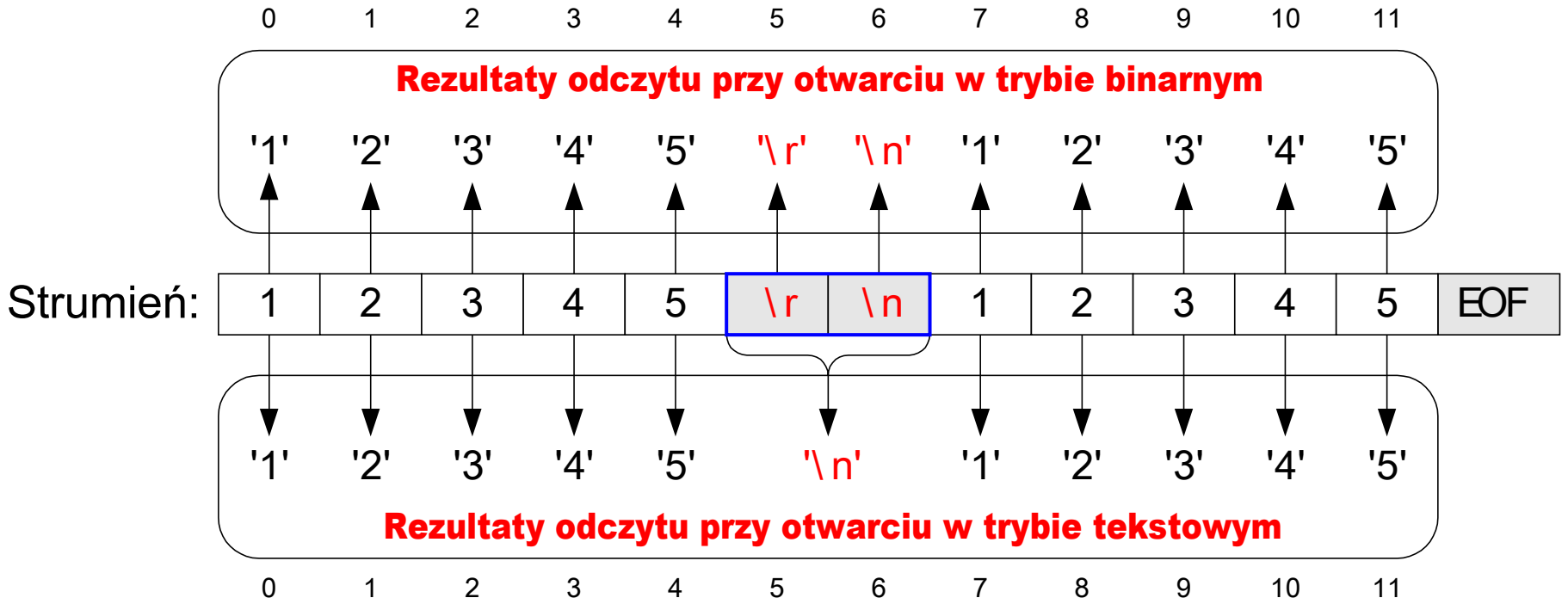
Spróbujmy wykorzystać podobną sekwencję dla wyznaczenia rozmiaru pliku liczonego w bajtach, tym razem wykorzystajmy iteracje *for*:

```
if( ( fp = fopen( "d.txt", "rt" ) ) != NULL )
{
  for( counter = 0; fgetc( fp ) != EOF; counter++  )
    ;
  fclose( fp );
  printf( "\nRozmiar pliku: %ld bajtow", counter );
}
```



Dlaczego rozmiary się nie zgadzają?

Przyczyną wadliwego działania programu są konwersje znaczników końca linii w trybie tekstowym. W systemach DOS/Windows znacznik końca linii to para *\r* i *\n* (czyli *CR* i *LF*). W trakcie odczytu w trybie tekstowym, każda para *\r\n* zamieniana jest na pojedynczy znak *\n*.



\r \n Znacznik końca wiersza

Otwarcie jako plik binarny gdy nie przetwarzamy tekstów

Konwersje nie zachodzą przy otwieraniu pliku w trybie binarnym. W drugim parametrze wywołania *fopen* należy użyć litery *b*, oznaczającej otwarcie w trybie binarnym. Poprawiona wersja kodu wkomponowana w funkcję *file_size*:

```
long int file_size( char * fname )
{
  FILE * fp;
  long int counter = 0;

  if( ( fp = fopen( fname, "rb" ) ) != NULL )
  {
    for( counter = 0; fgetc( fp ) != EOF; counter++ )
      ;
   fclose( fp );
  }
  return counter;
}
. . .
printf( "\nRozmiar pliku: %ld bajtow", file_size( "d.txt" ) );
```

Kopiowanie zawartości plików

Wykonaj kopię pliku *dane.txt* w pliku *dane.bak*:

```
FILE * src = NULL;
FILE * dst = NULL;

if( ( src = fopen( "dane.txt", "rb" ) ) != NULL )
{
  if( ( dst = fopen( "dane.bak", "wb" ) ) != NULL )
  {
    int c;

    while( ( c = fgetc( src ) ) != EOF )
      fputc( c, dst );

    fclose( dst );
  }
  else
    printf( "Blad tworzenia pliku kopii zapasowej" );

  fclose( src );
}
else
  printf( "Blad otwarcia pliku zrodlowego" );
```

Kopiowanie zawartości plików — funkcja cpc_file_copy

```
/*-------------------------------------------------------------
  Funkcja cpc_file_copy realizuje kopiowanie zawartości źródłowego
  pliku src do pliku docelowego dst. Wykorzystywane są znakowe
  operacje odczytu i zapisu. Funkcja nie zamyka strumieni src i
  dst.

  Prametry : Wskaźniki na prawidłowo otwarte strumienie binarne
             src, dst - odpowiednio dla pliku źródłowego i
             docelowego

  Rezultat : true  jeżeli kopiowanie zakonczyło się poprawnie
             false jeżeli wystapił błąd podczas kopiowania
-------------------------------------------------------------*/
bool cpc_file_copy( FILE * dst, FILE * src )
{
  int c;

  for( ; ( c = fgetc( src ) ) != EOF ; fputc( c, dst ) )
     if( ferror( src ) || ferror( dst ) )
       return false;
  return true;
}
```

Kopiowanie ze zmianą zawartości, przykład 1-szy

```
/*-------------------------------------------------------------
  Funkcja toupper_file_copy realizuje kopiowanie zawartości
  źrodłowego
  pliku src do pliku docelowego dst. W trakcie kopiowanie wszystkie
  litery małe są zamieniane na duże a tzw. ”biale spacje” na znak
  myślnika '-'. Wykorzystywane są znakowe operacje odczytu i zapisu.
  Funkcja nie zamyka strumieni src i dst.
  Parametry : Wskaźniki na prawidłowo otwarte strumienie binarne
              src, dst - odpowiednio dla pliku źrodlowego i
              docelowego
  Rezultat  : true  jeżeli kopiowanie zakonczyło się poprawnie
              false jeżeli wystapił błąd podczas kopiowania
-------------------------------------------------------------*/
bool toupper_file_copy( FILE * dst, FILE * src )
{
  int c;

  while( ( c = fgetc( src ) ) != EOF )
  {
    fputc( ( isspace( c ) ) ? '-' : toupper( c ), dst );
    if( ferror( src ) || ferror( dst ) )
      return false;
  }
  return true;
}
```

Kopiowanie ze zmianą zawartości, przykład 2-gi i 3-ci

W trakcie kopiowania można realizować filtrowanie znaków, np. kopiowanie tylko liter i cyfr:

```
while( ( c = fgetc( src ) ) != EOF )
{
  if( isalnum( c ) )
    fputc( c, dst );
}
```

Zamiana liter dużych na małe:

```
while( ( c = fgetc( src ) ) != EOF )
  fputc( tolower( c ), dst );
```

Zamiana liter małych na duże:

```
while( ( c = fgetc( src ) ) != EOF )
  fputc( toupper( c ), dst );
```

Przetwarzanie plików tekstowych linia po linii

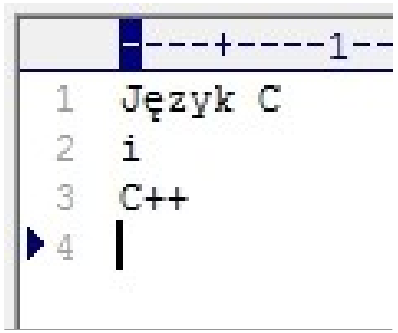
- Pliki tekstowe reprezentowane są również jako *strumienie bajtów*. Można je jednak *przetwarzać wierszami*, od strony programu separatorem wierszy jest znak *\n*.
- Do przetwarzania pliku tekstowego linia po linii służą funkcje odczytu/zapisu linii — buforem linii są *tablice znakowe*.

Zawartość pliku:

J	ę	z	y	k		C	\n	i	\n	C	+	+	\n	EOF

Przy przetwarzaniu *linia po linii* można założyć, że plik wygląda tak:

J	ę	z	y	k		C	\n
i	\n						
C	+	+	\n				
EOF							

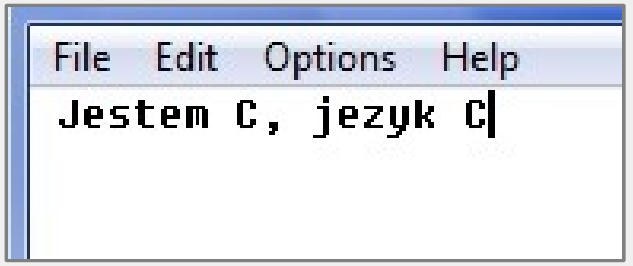


Przetwarzanie plików tekstowych linia po linii

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

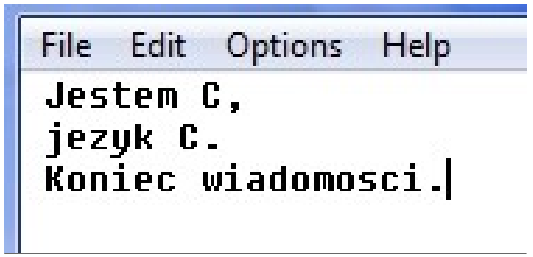
- Funkcja *fputs* wyprowadza napis *s* do pliku *stream*, *nie dopisuje* znaczników *końca wiersza* ani *końca napisu*. Rezultatem funkcji jest *ostatni zapisany znak*, w przypadku gdy zapis zakończył się sukcesem lub *EOF*, gdy wystąpił błąd.

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

int main()
{
  FILE * fp = NULL;

  if( ( fp = fopen( "dane.txt", "wt" ) ) != NULL )
  {
    fputs( "Jestem C", fp );
    fputs( ", jezyk C", fp );

    fclose( fp );
  }
  return EXIT_SUCCESS;
}
```



Wykorzystanie funkcji fputs

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

int main()
{
  FILE * fp = NULL;

  if( ( fp = fopen( "dane.txt", "wt" ) ) != NULL )
  {
    fputs( "Jestem C", fp );
    fputs( ",\njezyk C.\nKoniec wiadomosci.", fp );

   fclose( fp );
  }
  return EXIT_SUCCESS;
}
```



Funkcja fprintf

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

- Funkcja *fprintf* wyprowadza do pliku *stream* napis *format* oraz opcjonalne argumenty, w postaci określonej przez sekwencje formatujące zapisane w napisie *format*.
- Rezultatem funkcji jest *liczba wyprowadzonych* bajtów, w przypadku gdy zapis zakończył się sukcesem lub *EOF*, gdy wystąpił błąd..
- Wszystkie zasady formatowania znane z wykorzystania funkcji *printf* obowiązują dla funkcji *fprintf*.

Zamiast wywołania funkcji *printf*:

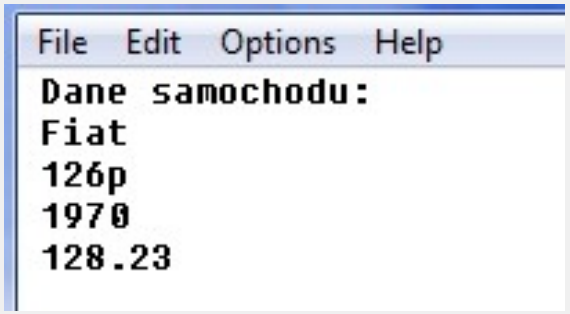
```
printf( "printf to fprintf piszacy do stdout" );
```

można napisać:

```
fprintf( stdout, "printf to fprintf piszacy do stdout" );
```

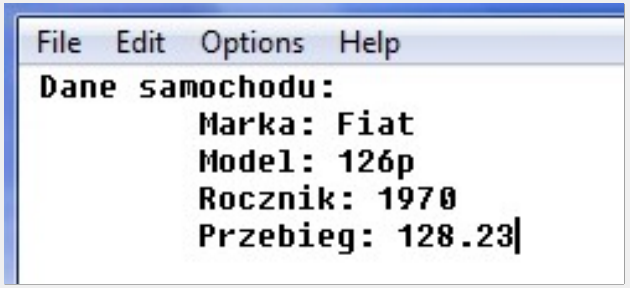
Wykorzystanie funkcji fprintf, wersja 1-sza

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

int main()
{
  FILE * fp = NULL;

  if( ( fp = fopen( "dane.txt", "wt" ) ) != NULL )
  {
    char marka[ 80 ] = "Fiat";
    char model[ 80 ] = "126p";
    int  rocznik = 1970;
    float przebieg = 128.23;

    fprintf( fp, "Dane samochodu:\n%s\n%s\n%d\n%g", marka, model,
             rocznik, przebieg );

    fclose( fp );
  }
  return EXIT_SUCCESS;
}
```



Wykorzystanie funkcji fprintf, wersja 2-ga

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

int main()
{
  FILE * fp = NULL;

  if( ( fp = fopen( "dane.txt", "wt" ) ) != NULL )
  {
    char marka[ 80 ] = "Fiat";
    char model[ 80 ] = "126p";
    int  rocznik = 1970;
    float przebieg = 128.23;

    fprintf( fp, "Dane samochodu:\n\tMarka: %s\n\tModel: %s\n", marka, model);
    fprintf( fp, "\tRocznik: %d\n\tPrzebieg: %g", rocznik, przebieg );

    fclose( fp );
  }
  return EXIT_SUCCESS;
}
```



Funkcja fgets

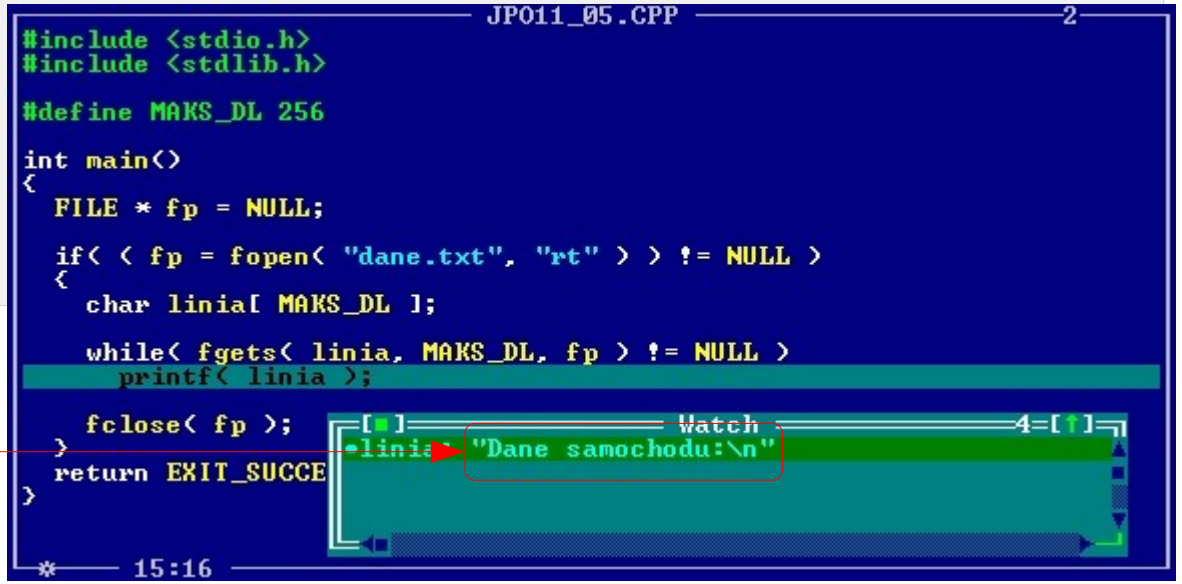
```
char * fgets( char * s, int n, FILE * stream );
```

- Pierwszy parametr *s* określa bufor, do którego mają być zapisane wczytywane dane.
- Drugi parametr *n* określa maksymalną pojemność bufora, uwzględniającą miejsce na znacznik końca napisu.
- Trzeci parametr *stream* określa strumień (plik), z którego funkcja ma odczytywać dane, może to być również standardowy strumień wejściowy – *stdin*.
- Działanie funkcji kończy się gdy funkcja odczyta $n - 1$ znaków lub wcześniej zostanie odczytany znak nowego wiersza (Enter).
- Znacznik końca napisu dopisywany jest na jego końcu.

Wykorzystanie funkcji fgets, uwaga na znacznik końca wiersza

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

#define MAKS_DL 256

int main()
{
  FILE * fp = NULL;

  if( ( fp = fopen( "dane.txt", "rt" ) ) != NULL )
  {
    char linia[ MAKS_DL ];

    while( fgets( linia, MAKS_DL, fp ) != NULL )
      printf( linia );

    fclose( fp );
  }
  return EXIT_SUCCESS;
}
```

```
Dane samochodu:
        Marka: Fiat
        Model: 126p
        Rocznik: 1970
        Przebieg: 128.23
```



Funkcja *fgets* pozostawia w buforze znacznik końca wiersza.

Wykorzystanie funkcji fgets, uwaga na znacznik końca wiersza

Uwaga, przed znacznikiem końca pliku *EOF* może nie być znacznika końca wiersza \n. Funkcja *fgets* nie przeczyta go zatem.

Zawartość pliku:

J	ę	z	y	k		C	\n	i	\n	C	+	+	EOF

Przy przetwarzaniu *linia po linii* można założyć, że plik wygląda tak:

J	ę	z	y	k		C	\n
i	\n						
C	+	+	EOF				


```
while( fgets( linia, MAKS_DL, fp ) != NULL )
    printf( linia );

fclose( fp );
}
return EXIT_SUCCE
```

Watch
linia: "C++"

Wykorzystanie fgets w funkcji list_file

```
/*-------------------------------------------------------------
  Wyprowadza do stdout zawartość pliku o nazwie fname.
  Parametry: char * fname – wskaźnik na tablicę zawierającą nazwę
             pliku.
  Rezultat:  Brak.
-------------------------------------------------------------*/
#define MAX_LINE 256
void list_file( char * fname )
{
  FILE * fp;
  char buffer[ MAX_LINE ];

  if( ( fp = fopen( fname, "rt" ) ) != NULL )
  {
    while( fgets( buffer, MAX_LINE, fp ) != NULL )
      printf( "%s", buffer );
   fclose( fp );
  }
}
```

```
. . .
list_file( "przyklad4.c" );
. . .
```

```
/*-------------------------------------------------------------
 Funkcja: long int nlist_file( char * fname )
   Wyprowadza do stdout zawartość pliku o nazwie fname, każda linia
   poprzedzona jest jej numerem.
 Rezultat:
   Liczba linii pliku o nazwie zapisane w fname
-------------------------------------------------------------*/
#define MAX_LINE 256
long int nlist_file( char * fname )
{
  FILE * fp;
  long int counter = 0;
  char buffer[ MAX_LINE ];

  if( ( fp = fopen( fname, "rt" ) ) != NULL )
  {
    while( fgets( buffer, MAX_LINE - 1, fp ) != NULL )
      printf( "%3ld: %s", ++counter, buffer );
   fclose( fp );
  }
  return counter;
}
```

Funkcja list_file zawija za długie wiersze

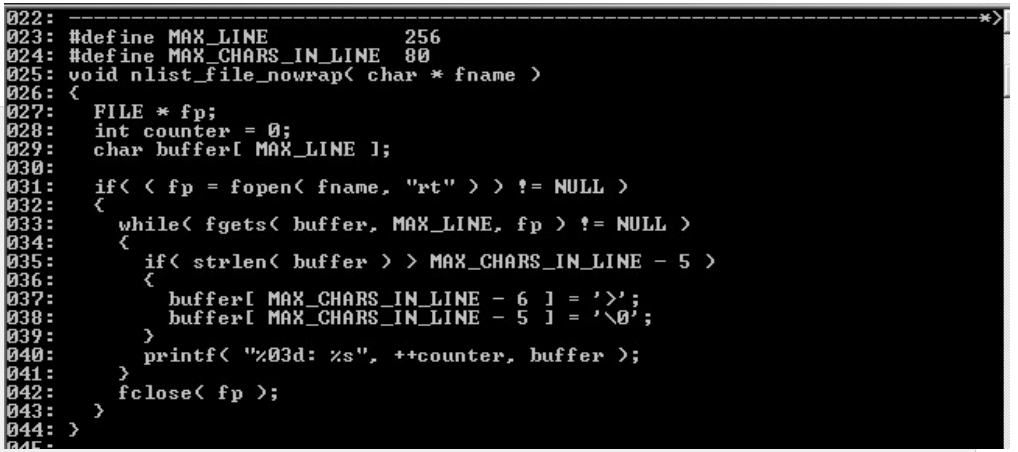
```
To jest plik tekstowy przeznaczony do testowania funkcji przetwarzajacych pliki,
 pisanych w jezyku C.
Ten plik zawiera dwie dlugie linie,sa one dluzsze od 80-ciu znakow, czyli zwycza
jowej szerokosci ekranu w trybie konsolowym.
Ta linia, i nastepna, sa krotkie.
Druga krotka linia.
```

Można zmodyfikować funkcję *list_file* tak, by nie łamała za długich wierszy a umieszczała na ich końcu symbol informujący, że wiersz jest dłuższy od szerokości ekranu.

Funkcja list_file_nowrap przycina za długie wiersze

```
#define MAX_LINE           256
#define MAX_CHARS_IN_LINE  80

void list_file_nowrap( char * fname )
{
  FILE * fp;
  char buffer[ MAX_LINE ];

  if( ( fp = fopen( fname, "rt" ) ) != NULL )
  {
    while( fgets( buffer, MAX_LINE, fp ) != NULL )
    {
      if( strlen( buffer ) > MAX_CHARS_IN_LINE )
      {
        buffer[ MAX_CHARS_IN_LINE - 1 ] = '>';
        buffer[ MAX_CHARS_IN_LINE ]     = '\0';
      }

      printf( "%s", buffer );
    }
    fclose( fp );
  }
}
```

Funkcja nlist_file_nowrap dodatkowo numeruje wiersze



```
#define MAX_LINE           256
#define MAX_CHARS_IN_LINE  80

void nlist_file_nowrap( char * fname )
{
  FILE * fp;
  int counter = 0;
  char buffer[ MAX_LINE ];

  if( ( fp = fopen( fname, "rt" ) ) != NULL )
  {
    while( fgets( buffer, MAX_LINE, fp ) != NULL )
    {
      if( strlen( buffer ) > MAX_CHARS_IN_LINE - 5 )
      {
        buffer[ MAX_CHARS_IN_LINE - 6 ] = '>';
        buffer[ MAX_CHARS_IN_LINE - 5 ] = '\0';
      }
      printf( "%03d: %s", ++counter, buffer );
    }
    fclose( fp );
  }
}
```

Funkcja pattern_list_file wyświetla linie z wzorcem

```
void pattern_list_file( char * fname, char * pattern )
{
  FILE * fp;
  int counter = 0;
  char buffer[ MAX_LINE ];

  if( ( fp = fopen( fname, "rt" ) ) != NULL )
  {
    while( fgets( buffer, MAX_LINE - 1, fp ) != NULL )
    {
      ++counter;
      if( strstr( buffer, pattern ) != NULL )
      {
        if( strlen( buffer ) > MAX_CHARS_IN_LINE - 5  )
        {
          buffer[ MAX_CHARS_IN_LINE - 6 ] = '>';
          buffer[ MAX_CHARS_IN_LINE - 5 ] = '\0';
        }
        printf( "%03ld: %s", counter, buffer );
      }
    }
    fclose( fp );
  }
}
```

```
032:   if( ( fp = fopen( fname, "rt" ) ) != NULL )
037:       if( strstr( buffer, pattern ) != NULL )
040:         if( strlen( buffer ) > 80  )
063:   if( ( file = fopen( file_name, "rt" ) ) != NULL )
```

```
pattern_list_file( "przyklad6.c" , "if" );
```

Strumienie standardowe jako otwarte pliki

W pliku *stdio.h* zdefiniowana są trzy wskaźniki przypisane trzem strumieniom, automatycznie otwieranym dla programu:

- standardowy strumień wyjściowy: *stdout*,
- standardowy strumień wejściowy: *stdin*,
- standardowy strumień wyjściowy dla błędów: *stderr*.

Wskaźniki *stdin*, *stdout* i *stderr* są typu *FILE **, można z nich korzystać jak z otwartych plików:

```
printf( "Witaj!" );
```

```
fprintf( stdout, "Witaj!" );
```

```
putchar( 'A' );
```

```
fputc( 'A', stdout );
```

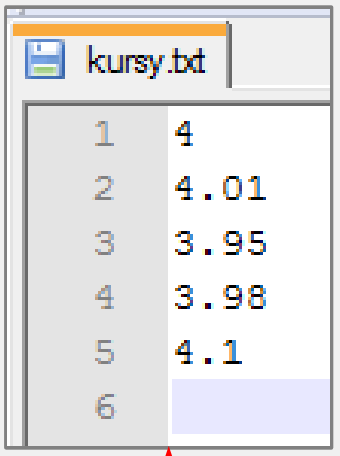
```
c = getchar();
```

```
c = fgetc( stdin );
```

Przykład 1: Dane z tablicy do pliku

Dana jest *N*-elementowa tablica liczb rzeczywistych *kursy*, zawierająca cenę zakupu waluty EURO wyrażoną w złotówkach. Jak zapisać zawartość tablicy do pliku tekstowego?

```
#include <cstdio>
#include <cstdlib>
using namespace std;

const int N = 5;
int main()
{
  // Tablica z przykładowymi danymi
  float kursyEURO[ N ] = { 4, 4.01, 3.95, 3.98, 4.1 };

  FILE * fp = NULL;

  if( ( fp = fopen( "kursy.txt", "wt" ) ) != NULL )
  {
    for( int i = 0; i < N; ++i )
      fprintf( fp, "%g\n", kursyEURO[ i ] );

    fclose( fp );
  }
  return EXIT_SUCCESS;
}
```



Przykład 2: Dane z pliku do tablicy

Jak odczytać dane z pliku tekstowego do tablicy kursów?

```
. . .
const int N = 5;
int main()
{
  // Tablica na dane
  float kursyEURO[ N ];

  FILE * fp = NULL;
  if( ( fp = fopen( "kursy.txt", "rt" ) ) != NULL )
  {
    int i = 0;

    while( fscanf( fp, "%g", &kursyEURO[ i++ ] ) == 1 )
      if( i == N )
        break;

    for( i = 0; i < N; ++i )
      printf( "\n%g", kursyEURO[ i ] );

    fclose( fp );
  }

  return EXIT_SUCCESS;
}
```

Zabezpieczenie przed przekroczeniem zakresu tablicy

Przykład 2: Dane z pliku do tablicy — funkcja fscanf

Funkcja *fscanf* to przedstawiciel rodziny funkcji (*scanf*, *sscanf*, ...) realizujących pobieranie danych („skanowanie”) z *pewnego źródła* i zapisanie ich do zmiennych programu zgodnie z zadanym formatem. Źródłem danych dla funkcji *fscanf* jest uprzednio otwarty plik.

Funkcja *fscanf* jest:

- bardzo *użyteczna*,
- posiada szereg ciekawych *możliwości*,
- wymaga *uwagi* i *przemyślanego* stosowania,
- stosowana nieuważnie jest *kapryśna* i *niebezpieczna*.

Bardzo interesujący opis wykorzystania funkcji *fscanf* zawiera książka:

Adam Sapek, *Wgłąb języka C*, Helion, Gliwice, 1993

Przykład 2: Dane z pliku do tablicy — funkcja fscanf

```
fscanf( FILE * file, const char * format );
```

Funkcja *fscanf* posiada dwa obowiązkowe parametery:

- `FILE * file` — wskaźnik pliku otwartego do odczytu,
- `const char * format` — ciąg znaków sterujący odczytem i formatowaniem danych (zwanym dalej *łańcuchem sterującym odczytem*),
- kolejne parametry *muszą być wskaźnikami na zmienne*, do których zostaną zapisane dane odczytane z pliku zgodnie z informacjami formatującymi zapisanymi w `format`.

Rezultatem funkcji *fscanf* jest liczba *przeczytanych*, *sformatowanych* i *zapamiętanych* danych (dane niezapamiętane *nie są zliczane*!).

W przypadku napotkania końca pliku przed zakończeniem odczytu rezultatem funkcji jest EOF.

Przykład 2: Dane z pliku do tablicy — funkcja fscanf

Odczytaj z pliku *fp* liczbę rzeczywistą, potraktują ją jako daną typu *float* i zapisz do zmiennej wskazywanej przez *&num*.

```
float num;

fscanf( fp, "%g", &num );
```

Odczytaj z pliku *fp* liczbę rzeczywistą, potraktują ją jako daną typu *double* i zapisz do zmiennej wskazywanej przez *&num* — uwaga na znak *l* poprzedzający *f*.

```
double num;

fscanf( fp, "%lg", &num );
```

Odczytaj z pliku *fp* liczbę rzeczywistą, potraktują ją jako daną typu *double* i zapisz do elementu tablicy wskazywanego przez *&kursyEURO[i]*.

```
float kursyEURO[ N ];

fscanf( fp, "%g", &kursyEURO[ i ] );
```

Przykład 2: Dane z pliku do tablicy — funkcja fscanf

Powtarzaj dopóki udaje się z pliku odczytać, sformatować i zapisać jedną liczbę rzeczywistą:

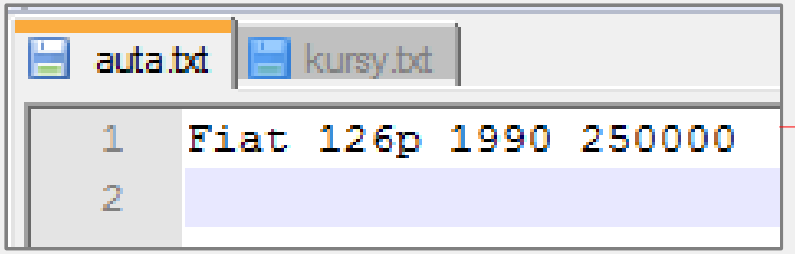
```
while( fscanf( fp, "%g", &kursyEURO[ i++ ] ) == 1 )
{
    . . .
}
```

Inny zapis, podobne działanie

```
while( fscanf( fp, "%g", &kursyEURO[ i++ ] ) != EOF )
{
  . . .
}
```

Przykład 3: Odczyt kilku elementów w pojedynczym wywołaniu fscanf

```
#include <cstdio>
#include <cstdlib>
using namespace std;

const int N = 20;
int main()
{
  char  marka[ N ];
  char  model[ N ];
  int   rokProd;
  float przebieg;

  FILE * fp = NULL;
  if( ( fp = fopen( "auta.txt", "rt" ) ) != NULL )
  {
    fscanf( fp, "%s %s %d %g", marka, model, &rokProd, &przebieg );

    printf( "\nMarka: %s\nModel: %s\nRocznik: %d\nPrzebieg: %g",
            marka, model, rokProd, przebieg );

    fclose( fp );
  }

  return EXIT_SUCCESS;
}
```




```
Marka: Fiat
Model: 126p
Rocznik: 1990
Przebieg: 250000
```

Przykład 3: Odczyt kilku elementów w pojedynczym wywołaniu fscanf

Ponieważ nazwy tablic są interpretowane jako wskaźniki na ich pierwsze elementy, w przypadku tablic nie stosujemy operatora &:

```
fscanf( fp, "%s %s %d %g", marka, model, &rokProd, &przebieg );
```

Zawsze jednak można wykorzystać zapis:

```
fscanf( fp, "%s %s %d %g", &marka[ 0 ], &model[ 0 ], &rokProd, &przebieg );
```

- Kolejne wczytywane elementy rozdzielane są tzw. „białymi znakami” (spacja, tabulacja, przejście do nowego wiersza).
- Jeżeli napotkany znak *nie pasuje* do specyfikacji określonej w łańcuchu sterującym odczytem, działanie funkcji *fscanf* jest *przerywane*.
- Specyfikacja %s zakłada, że pole wejściowe jest ograniczone białymi znakami, co *nie pozwala* na wczytanie łańcucha znaków *zawierającego odstępy*.

Przykład 4: Jak Przykład 3 tylko wiele razy

```
#include <cstdio>
#include <cstdlib>
using namespace std;

const int N = 20;
int main()
{
  char  marka[ N ];
  char  model[ N ];
  int   rokProd;
  float przebieg;

  FILE * fp = NULL;
  if( ( fp = fopen( "auta.txt", "rt" ) ) != NULL )
  {
    while( fscanf(fp, "%s %s %d %g", marka,model,&rokProd,&przebieg) != EOF )
      printf( "\nMarka: %s\nModel: %s\nRocznik: %d\nPrzebieg: %g\n",
              marka, model, rokProd, przebieg );

    fclose( fp );
  }

  return EXIT_SUCCESS;
}
```

auta.txt | kursy.txt

```
1  Fiat 126p 1990 250000
2  Ford T 1908 250
3  Syrena Bosto 1966 55000
```

```
Marka: Fiat
Model: 126p
Rocznik: 1990
Przebieg: 250000

Marka: Ford
Model: T
Rocznik: 1908
Przebieg: 250

Marka: Syrena
Model: Bosto
Rocznik: 1966
Przebieg: 55000
```

Przykład 5: Zapis danych z rekordu do pliku tekstowego

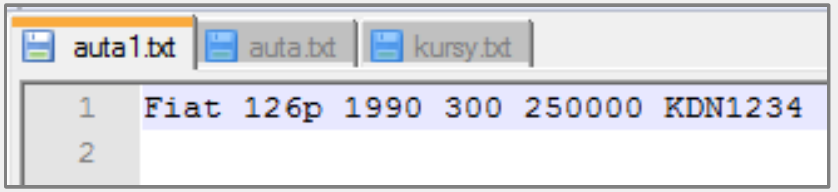
```
#include <cstdio>
#include <cstdlib>
using namespace std;

const int MAKS_M = 20;
const int MAKS_R = 10;

struct pojazd
{
  char  marka[ MAKS_M ];
  char  model[ MAKS_M ];
  int   rok_prod;
  float cena;
  float przebieg;
  char  nr_rej[ MAKS_R ];
};
```

Przykład 5: Zapis danych z rekordu do pliku tekstowego

```
int main()
{
  pojazd p = { "Syrena", "Bosto", 1966, 1, 12000, "KTA1234" };

  FILE * fp = NULL;
  if( ( fp = fopen( "auta2.txt", "wt" ) ) != NULL )
  {
    fprintf( fp, "%s %s %d %g %g %s\n",
             p.marka, p.model, p.rok_prod, p.cena, p.przebieg, p.nr_rej  );
    fclose( fp );
  }

  return EXIT_SUCCESS;
}
```

Przykład 6: Odczyt danych z pliku tekstowego do rekordu

```
int main()
{
  pojazd p;

  FILE * fp = NULL;
  if( ( fp = fopen( "auta1.txt", "rt" ) ) != NULL )
  {
    fscanf( fp, "%s %s %d %g %g %s", p.marka, p.model, &p.rok_prod,
            &p.cena, &p.przebieg, p.nr_rej );

    printf( "\nMarka: %s\nModel: %s\nRok: %d\nCena: %g"
            "\nPrzebieg: %g\nNr rej.: %s",
            p.marka, p.model, p.rok_prod, p.cena, p.przebieg, p.nr_rej  );

    fclose( fp );
  }

  return EXIT_SUCCESS;
}
```




```
Marka: Fiat
Model: 126p
Rok: 1990
Cena: 300
Przebieg: 250000
Nr rej.: KDN1234_
```

Przy okazji: dwa sąsiadujące ze sobą napisy, niczym nie oddzielone, są przez kompilator łączone w jeden napis, np.:

`"C" "++"` łączone jest w `"C++"`

Przykład 7: Jak przykład 6 tylko z pominięciem wybranych danych

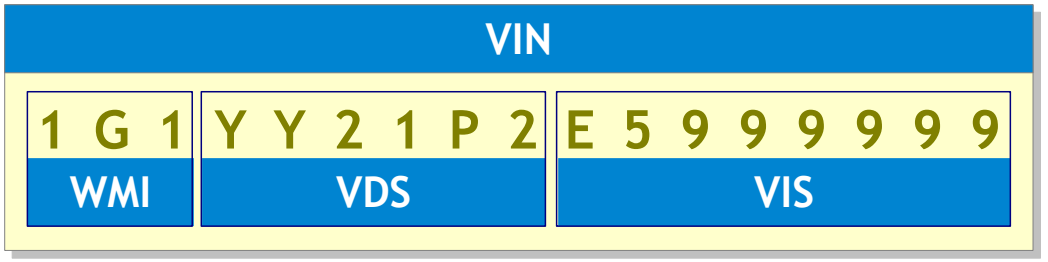
```
. . .
int main()
{
  pojazd p;

  FILE * fp = NULL;
  if( ( fp = fopen( "auta1.txt", "rt" ) ) != NULL )
  {
    // Czytaj z pominieciem roku produkcji i numeru rejestracyjnego

    fscanf( fp, "%s %s %*d %g %g %*s",
            p.marka, p.model, &p.cena, &p.przebieg );

    printf( "\nMarka: %s\nModel: %s\nCena: %g\nPrzebieg: %g\n",
            p.marka, p.model, p.cena, p.przebieg );

    fclose( fp );
  }

  return EXIT_SUCCESS;
}
```

Umieszczenie znaku * na początku specyfikacji formatu powoduje pominięcie danych.

Dokładniej – dane są *odczytywane* lecz *ignorowane*.

Przykład 8: Odczyt danych o zadanej szerokości pola

- **VIN** (**V**ehicle **I**dentification **N**umber) jest złożonym zestawem znaków, który zostaje nadany pojazdowi przez producenta w celu jego identyfikacji (istnieje norma ISO 3779 - 1983, która określa treść i budowę numeru identyfikacyjnego pojazdu).

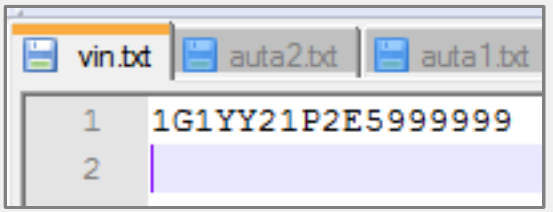


- **VIN** składa się z:
  - **WMI** — 3 znaki, *światowy symbol producenta*: rejon geograficzny, kraj, producent,
  - **VDS** — 6 znaków, *część opisująca pojazd*: konstrukcję samochodu, rodzaj nadwozia, rodzaj i odmianę silnika , układ przeniesienia napęd, kolejność i znaczenie określane są przez producenta.
  - **VIS** — 8 znaków, *część wyróżniająca pojazd* : identyfikuje dany egzemplarz samochodu, powinna zawierać numer fabryczny pojazdu. W części tej pierwsze cztery znaki są literowo-cyfrowe, a pozostałe cztery muszą być cyfrowe.

Przykład 8: Odczyt danych o zadanej szerokości pola

```
. . .
const int WMI_DL = 3;
const int VDS_DL = 6;
const int VIS_DL = 8;

struct VIN
{
  char  wmi[ WMI_DL + 1 ];
  char  vds[ VDS_DL + 1 ];
  char  vis[ VIS_DL + 1 ];
};

int main()
{
  VIN id;

  FILE * fp = NULL;
  if( ( fp = fopen( "vin.txt", "rt" ) ) != NULL )
  {
    fscanf( fp, "%3s%6s%8s", id.wmi, id.vds, id.vis );

    printf( "\nVIN: %s %s %s\nWMI: %s\nVDI: %s\nVIS: %s\n",
            id.wmi, id.vds, id.vis, id.wmi, id.vds, id.vis  );
    fclose( fp );
  }
  return EXIT_SUCCESS;
}
```

Określenie szerokości wczytywanego pola, liczone w znakach.

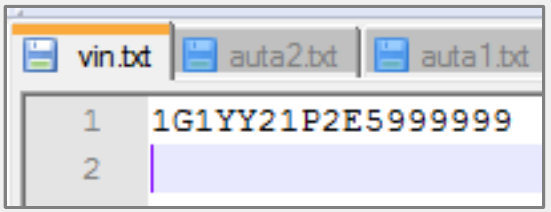



```
VIN: 1G1 YY21P2 E5999999
WMI: 1G1
VDI: YY21P2
VIS: E5999999
```

Przykład 9: Jak przykład 8 tylko z wykorzystaniem sscanf

```
. . .
const int STR_DL = 128;

struct VIN
{
  char  wmi[ WMI_DL + 1 ];
  char  vds[ VDS_DL + 1 ];
  char  vis[ VIS_DL + 1 ];
};

int main()
{
  VIN id;
  char linia[ STR_DL ];

  FILE * fp = NULL;
  if( ( fp = fopen( "vin.txt", "rt" ) ) != NULL )
  {
    fgets( linia, STR_DL, fp );
    sscanf( linia, "%3s%6s%8s", id.wmi, id.vds, id.vis );

    printf( "\nVIN: %s %s %s\nWMI: %s\nVDI: %s\nVIS: %s\n",
            id.wmi, id.vds, id.vis, id.wmi, id.vds, id.vis  );
    fclose( fp );
  }
  return EXIT_SUCCESS;
}
```

Wczytaj linię z pliku, zapisz w tablicy znaków, a następnie wczytaj elementy numeru VIN z tablicy znaków.



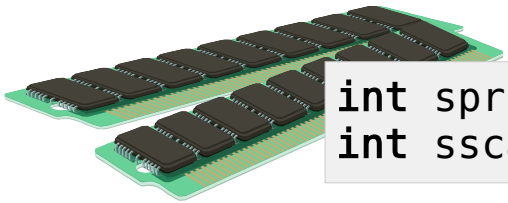

```
VIN: 1G1 YY21P2 E5999999
WMI: 1G1
VDI: YY21P2
VIS: E5999999
```

Przykład 9: Rodzina funkcji printf i scanf

- Funkcje *printf* i *scanf* występują w kilku odmianach.
- Działają zwykle tak samo, różnią się *miejscem*:
  - do którego *zapisują wyniki* swego działania — rodzina *printf*,
  - z którego *odczytują dane* — rodzina *scanf*.


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

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



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

Przykład 9: Funkcje sprintf i sscanf

- Funkcja *sprintf* pobiera zawartość kolejnych pól rekordu *id*, formatuje zgodnie z ustalonym formatem, zapisuje do tablicy znaków *linia*, funkcja *puts* wyprowadza zawartość tej tablicy do *stdout*.

```
VIN id = { "1G1", "YY21P2", "E5999999" };
char linia[ STR_DL ];

sprintf( linia, "%3s:%6s:%8s", id.wmi, id.vds, id.vis );
puts( linia );
```

```
1G1:YY21P2:E5999999
```

- Funkcja *gets* odczytuje ciąg znaków z *stdin*, zapisuje do tablicy *linia*, *sscanf* traktuje tę tablicę jako źródło danych, odczytuje z niej informacje zgodnie z zawartością łańcucha formatującego, zapisuje je do pól rekordu *id*, funkcja *printf* wyprowadza zawartość kolejnych pól tego rekordu do *stdout* zgodnie z ustalonym formatem.

```
VIN id;
char linia[ STR_DL ];

printf( "\nPodaj VIN: " );
gets( linia );
sscanf( linia, "%3s%6s%8s", id.wmi, id.vds, id.vis );
printf( "%3s:%6s:%8s", id.wmi, id.vds, id.vis );
```

```
Podaj VIN: 1G1YY21P2E5999999
1G1:YY21P2:E5999999
```

Przykład 10: Odczyt danych rozdzielonych separatorem

- Gdy wczytywane napisy mogą zawierać białe znaki — np. spacje — wczytywanie specyfikatorem **`%s`** staje się niemożliwe, ponieważ zatrzymuje on wczytywanie po napotkaniu białego znaku.

- Do wczytywania takich napisów służy specyfikacja **`%[]`**. Wewnątrz nawiasów zapisuje się ciąg *akceptowanych znaków*, każdy inny znak traktowany jest jako *ogranicznik pola*, jego napotkanie *kończy* odczyt elementu.

- Odczyt pliku *fp* do tablicy znaków *s* napis, który zawierać może tylko litery ABC, znaki podkreślenia '_' i spacje (np:. **`A_B_C_AB_AC_BC ABC`**):

```
fscanf( fp, "%[ABC_ ]", s );
```

- Odczyt pliku *fp* do tablicy znaków *s* napis, który zawierać może tylko małe litery, cyfry, i znaki podkreślenia '_', '-' i spacje (np.: **`007 james_bond j23 hans_kloss`**):

```
fscanf( fp, "%[a-z0-9_ -]", s );
```

`ABCDEF...Z` zastępuje **`A-Z`**, **`abcdef...z`** zastępuje **`a-z`**, **`0123456789`** zastępuje **`0-9`**

Przykład 10: Odczyt danych rozdzielonych separatorem

- W specyfikacji **`%[znaki]`** ogranicznikiem wczytywanego pola jest każdy znak *nie wymieniony* w nawiasach.
- W specyfikacji **`%[^znaki]`** ogranicznikiem wczytywanego pola jest każdy znak *wymieniony* w nawiasach.
- Czytaj tylko duże litery — ogranicznikiem pola jest każdy znak *nie będący* dużą literą:

```
fscanf( fp, "%[A-Z]", s );
```

- Czytaj wszystko aż do dużej litery — ogranicznikiem pola jest duża litera:

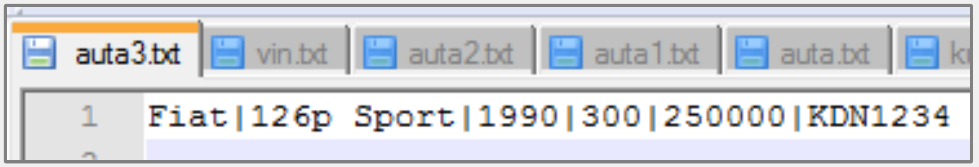
```
fscanf( fp, "%[^A-Z]", s );
```

- Zazwyczaj w nawiasach umieszcza się ogranicznika pola (separatory), czytaj wszystko aż do znaku '|', pomiń go:

```
fscanf( fp, "%[^|]%*c", s );
```

Przykład 10: Odczyt danych rozdzielonych separatorem

Kolejne informacje o samochodzie rozdzielane są znakami '|', znak ten jest *separatorem*. Dzięki temu marka i model samochodu mogą zawierać spacje. Uwaga — separator należy *przeczytać* i *pominąć*, stąd `%*c`, ale tylko dla specyfikacji `%[]`.




```
. . .

int main()
{
  pojazd p;

  FILE * fp = NULL;
  if( ( fp = fopen( "auta3.txt", "rt" ) ) != NULL )
  {
    fscanf( fp, "%[^|]%*c%[^|]%*c%d%*c%g%*c%g%*c%s",
             p.marka, p.model, &p.rok_prod, &p.cena, &p.przebieg, p.nr_rej  );

    printf( "\nMarka: %s\nModel: %s\nRok: %d\nCena: %g"
            "\nPrzebieg: %g\nNr rej.: %s",
            p.marka, p.model, p.rok_prod, p.cena, p.przebieg, p.nr_rej  );

    fclose( fp );
  }
  return EXIT_SUCCESS;
}
```

Funkcja fscanf — podsumowanie

- Uwaga, specyfikacja **`%[znaki]`** nie pomija białych znaków, zatem gdy pole ma zawartość: `|  123|`, dane nie zostaną odczytane, gdyż spacje z początku pola nie pasują do wzorca:

```
fscanf( fp, "%[0-9]", s );  // s == "?"
```

- Specyfikacja **`%s`** pomija białe znaki, zatem pole `|  123|`, zostanie prawidłowo odczytane:

```
fscanf( fp, "%s", s ); // s == "123"
```

- Funkcja *fscanf* potrafi naprawdę dużo (np. odczytywać liczby szesnastkowe, ósemkowe, uwzględniać ograniczenia szerokości pola, itp.), potrafi też również naprawdę zaskakiwać — wymaga uwagi i myślenia. Więcej informacji:
  - http://www.cplusplus.com/reference/clibrary/cstdio/fscanf/
  - http://pubs.opengroup.org/onlinepubs/009695399/functions/fscanf.html
  - http://www.kernel.org/doc/man-pages/online/pages/man3/scanf.3.html