

Języki i metody programowania I

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

<http://home.agh.edu.pl/~pszwed/>

Aktualizacja: 2013-01-25

11. Biblioteka wejścia

wyjścia

Wstęp

- Mechanizmy IO (wejścia / wyjścia) nie są częścią języka. W języku C wykorzystuje się jednak zestaw standardowych funkcji IO, które w postaci standardowej biblioteki `stdio` występują praktycznie we wszystkich implementacjach języka. (Nagłówek `<stdio.h>`)
- Zadaniem funkcji biblioteki `stdio` jest komunikacja z urządzeniami zewnętrznymi (systemem plików, klawiaturą, ekranem konsoli, drukarkami, modemami, itd.)
- Zestaw funkcji został tak dobrany, aby pokryć większość typowych operacji wykonywanych podczas komunikacji z urządzeniami zewnętrznymi.
- Jeżeli w programach nie będą używane żadne mechanizmy specyficzne dla architektury sprzętowej, zapewniona zostanie przenośność na poziomie kodu źródłowego.

Pliki (1)

Pliki

Podstawowymi obiektami, którymi manipulują funkcje biblioteki *stdio* są pliki. Takie podejście jest zgodne z konstrukcją systemu UNIX, dla którego język C został stworzony.

W systemie UNIX występują trzy rodzaje plików:

- pliki zwykłe
- pliki specjalne
- katalogi

Pliki zwykłe są zapisywane na dysku i zawierają dane lub programy. Pliki są traktowane jako ciągi bajtów. System operacyjny przechowuje informacje o nazwie, wielkości, atrybutach i lokalizacji pliku. Pozwala także na swobodny dostęp do dowolnego bajta pliku.

Programowo można realizować dostęp sekwencyjny (odczyt kolejnych danych) lub dostęp swobodny (ang. *random access*)

Pliki (2)

Pliki specjalne W systemie UNIX wszystkie urządzenia IO są widziane jako pliki. Mają one konkretne nazwy i są umieszczane w odpowiednich katalogach.

- Np.: `/dev/lp` jest nazwą pliku związanego ze standardową drukarką systemową. Skopiowanie pliku do `/dev/lp` nie spowoduje umieszczenie tam jego zawartości, ale uaktywnienie drukarki i wydruk.

Podobnie w systemie DOS:

- LPT1 oznacza drukarkę przyłączoną do portu równoległego.
- CON konsolę
- COM2 urządzenie przyłączone do portu szeregowego

Polecenie:

```
copy opis.txt lpt1
```

spowoduje wydruk pliku opis.txt na drukarce

Pliki (3)

Zapis na konsolę pod DOS (również Windows7)...

```
int main()
{
    FILE*file;
    file=fopen("con","wt");
    if(!file)return;
    fprintf(file,"Ala ma kota\n");
    fclose(file);
    return 0;
}
```

Ala ma kota

Pliki binarne i tekstowe (1)

- Pliki tekstowe zawierają znaki drukowalne, stąd w zasadzie powinny zawierać kody ASCII < 128 . W praktyce znaki z zakresu [128-255] również mają reprezentację graficzną.
- Pliki tekstowe są często przesyłane pomiędzy urządzeniami. Urządzenia typu drukarka, konsola realizują wyłączenie sekwencyjny dostęp do pliku, stąd konieczny jest wybór znaku oznaczającego koniec pliku.
- Znakiem tym jest CTRL-D (kod 4) dla systemu UNIX lub CTRL-Z dla DOS/Windows (kod 26). Umieszczenie takiego znaku w pliku tekstowym powoduje, że dalsza jego część może zostać zignorowana.

Pliki binarne i tekstowe (2)

- Znacznikiem końca wiersza w systemie UNIX jest ‘\n’ (kod 10). W systemie DOS/Windows para znaków ‘\r\n’ (kody 13 i 10). Pliki tekstowe nie są więc bezpośrednio przenośne pomiędzy systemami.
- Otwierając plik w systemie DOS/Windows określamy tryb dostępu jako **binarny** lub **tekstowy**.
- Podczas odczytu i zapisu w trybie **tekstowym** następuje automatyczna translacja znaków. Fizycznie występująca para znaków ‘\r\n’ zamieniana jest przy odczycie na ‘\n’. Zapis znaku ‘\n’ powoduje translację na ‘\r\n’.
- W systemie UNIX rozróżnienie pomiędzy trybami nie występuje.

Struktura FILE

Biblioteka *stdio* jest zaprojektowana wykorzystując podejście nazywane *abstrakcyjnymi typami danych*.

- Zdefiniowana w bibliotece struktura `FILE` opisuje plik: jego tryb otwarcia (odczyt, zapis), położenie kursora wskazującego bieżące miejsce w pliku, bufor danych pliku.
- Programista **nigdy** nie ma potrzeby bezpośredniego dostępu do pól struktury `FILE`. Zamiast tego posługuje się jedną z funkcji biblioteki *stdio*, której parametrem jest wskaźnik do struktury `FILE` związanej z danym plikiem.

Programy korzystające z biblioteki *stdio* automatycznie uzyskują dostęp do trzech standardowych plików otwartych w trybie tekstowym identyfikowanych przez zmienne typu `FILE*`.

<code>stdin</code>	Standardowe wejście (z klawiatury). Odpowiednikiem w C++ jest strumień <code>cin</code> .
<code>stdout</code>	Standardowe wyjście (ekran). Odpowiednikiem w C++ jest strumień <code>cout</code> .
<code>stderr</code>	Standardowy strumień błędów (ekran). Odpowiednikiem w C++ jest strumień <code>cerr</code> .

Inne pliki powinny być jawnie otwierane i zamykane !

Funkcje fopen i fclose (1)

Funkcje te umożliwiają otwarcie i zamknięcie pliku.

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

Otwiera plik. Zwraca wskaźnik do struktury `FILE` identyfikującej plik lub `0 (NULL)` w przypadku błędu.

<code>filename</code>	Nazwa pliku. Może zawierać pełną ścieżkę, np: "c:\\mojepliki\\plik.txt"
<code>mode</code>	Łańcuch tekstowy określający tryb dostępu r – read r+ – read & write w – write w+ – write & read a – append b – binary t – text

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

Zamyka plik `stream`.

Funkcje fopen i fclose (2)

Przykład:

```
FILE*file;  
file=fopen("c:\\mojepliki\\plik.txt","rt");  
if(!file){/* obsługa błędu */ }  
//...  
fclose(file);
```

Dlaczego nie można otworzyć pliku?

- Plik otwarty do odczytu (tryb „r”) nie istnieje
- Otwieramy plik do zapisu w nieistniejącym katalogu
- Otwieramy plik do zapisu na urządzeniu *read-only* (CD ROM) lub w katalogu, do którego nie mamy praw zapisu.
- Otwieramy plik do zapisu. Plik o danej nazwie istnieje i ma atrybut *read-only*
- Otwieramy plik do zapisu. Plik o danej nazwie istnieje i jest otwarty przez inną aplikację w trybie uniemożliwiającym zapis.

Funkcje `getc` i `putc`

Deklaracje

```
int getc( FILE *stream );  
int putc( int c, FILE *stream );
```

Funkcja `getc` odczytuje pojedynczy znak ze strumienia, funkcja `putc` wstawia znak do strumienia. Zwracana wartość `EOF` (`-1`) jest wskaźnikiem błędu.

```
int main()  
{  
    int c;  
    for(;;){  
        c = getc(stdin); // getchar()  
        if(c==EOF)break;  
        c = toupper(c);  
        putc(c,stdout); // putchar(c)  
    }  
}
```

Wywołanie:

```
toUpper.exe [należy wprowadzać linie, koniec po ^Z ]  
type in.txt | toUpper.exe  
toUpper.exe < in.txt > out.txt
```

Uwaga: w przypadku przekierowania wyjścia z programu do pliku, komunikaty wypisywane do strumienia `stderr` będą pojawiały się na konsoli.

Funkcja fprintf (1)

Funkcja `fprintf()` umożliwia sformatowanie argumentów według zadanego wzorca i wypisanie do pliku.

Deklaracja:

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

Funkcja `printf` może być traktowana jako wywołanie:

```
fprintf(stdout, format [, argumenty]);
```

Funkcja fprintf (2)

Format

Tablica znakowa format określa sposób formatowania argumentów. Każdy argument formatowany jest zgodnie ze wzorcem postaci:

```
%[flags] [width] [.precision] type
```

- `type` – określa typ argumentu: znakowy (`c`), całkowitoliczbowy (`d`, `u`, `o`, `x`), zmiennoprzecinkowy (`e`, `f`, `g`), łańcuch znaków `s`.
- `flags` – sterują wyrównaniem i wypełnianiem pustych pól
- `width` – określa minimalną liczbę znaków wypisywanych przy wydruku; brakujące pola mogą być uzupełniane zgodnie ze specyfikacją `flags` (np.: spacjami)
- `precision` – Określa maksymalną liczbę wypisanych znaków dla formatów całkowitoliczbowych, maksymalną liczbę miejsc po kropce dziesiętnej dla formatów zmiennoprzecinkowych, maksymalną liczbę znaków dla łańcuchów znakowych.

```
printf( "%.0d ", 0 )           || pusty tekst
printf( "%5.3d\n", 1 )       |  001 |
printf( "%-5.3d\n", 12 )    |012  |
printf( "%+5.3d\n", 12 )    | +012 |
printf( "%-+5.3d\n", 12 )  |+012 |
```

Funkcja fscanf (1)

Deklaracja

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

Funkcja umożliwia odczyt argumentów z pliku tekstowego. Konstruowana jest jako komplementarna do funkcji fprintf().

Różnice

- Parametr format określa sposób synchronizacji z ciągiem znaków pojawiających się na wejściu.
 - Biały znak jest synchronizowany z dowolną liczbą białych znaków.
 - Niebiały znak, który nie należy do specyfikacji formatu uzgadniany jest z identycznym znakiem na wejściu. W przypadku braku zgodności funkcja kończy działanie.
- Argumentami funkcji `scanf` są **adresy** zmiennych (l-wartości) zdolnych przechować odczytaną wartość.
- Po dokonaniu konwersji należy sprawdzić wartość zwracaną przez `scanf` (liczbę przetworzonych pól).

Funkcja fscanf (2)

Przykład:

```
int x;  
float y ; // nie double!  
char z[256] ; // nie char* z; !!!  
  
fscanf(fin, "ABC %d %f %s", &x,&y,z ); // nie &z  
fprintf(fout, "ABC %d %f %s", x,y,z );
```

Uwagi:

Zarówno w przypadku funkcji fprintf (printf) oraz fscanf (scanf) istnieją ich odpowiedniki działające na tablicach znakowych.

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

Funkcje o zmiennej liczbie argumentów (1)

Funkcje `fprintf` i `fscanf` są przykładami funkcji, które można wywołać ze zmienną liczbą argumentów.

- Funkcje takie można konstruować dzięki specyficznej konwencji wołania funkcji w języku C:
- Argumenty wywołania funkcji `foo(arg1, arg2, ..., argn)` zapisywane są na stos w odwrotnej kolejności – najpierw: `argn`, na końcu: `arg1` i adres powrotu.
- Po przekazaniu sterowania do funkcji `foo` argumenty na stosie odwzorowywane w formalną listę parametrów (posługujemy się wartościami odłożonymi na stosie za pośrednictwem identyfikatorów).
- Na zakończenie argumenty są usuwane ze stosu. Standardowo dokonuje tego funkcja, z której wywołano `foo()`.
[Dla konwencji `__cdecl`]

Funkcje o zmiennej liczbie argumentów (2)

```
char format[] = "%s %s\n";  
char hello[] = "Hello";  
char world[] = "world";
```

```
void main( void )  
{
```

```
Hello world  
12
```

```
    int result;
```

```
    __asm {
```

```
        mov  eax, offset world
```

```
        push eax
```

```
        mov  eax, offset hello
```

```
        push eax
```

```
        mov  eax, offset format
```

```
        push eax
```

```
        call printf
```

```
        //clean up the stack, use the unused register ebx
```

```
        pop  ebx
```

```
        pop  ebx
```

```
        pop  ebx
```

```
        mov  result, eax
```

```
    }
```

```
    printf("%d\n",result);
```

```
}
```

Algorytm dla funkcji printf

1. Po wywołaniu znany jest format (wskaźnik do łańcucha tekstowego).
2. Znajdź specyfikację formatowania %...type w łańcuchu tekstowym format. Jeśli takiej brak – STOP.
3. Odczytaj kolejną wartość na stosie
4. Na podstawie znaku type odwzoruj w zmienną określonego typu
5. Wypisz zmienną i przejdź do 2.

Funkcje o zmiennej liczbie argumentów (4)

Programową obsługę funkcji o zmiennej liczbie argumentów zapewniają stosunkowo proste makra preprocesora zdefiniowane w `<stdarg.h>`

```
int foo(int first,...){
    int i;
    double d ;
    va_list marker;
    // kopiuj adres first do marker
    va_start( marker, first );
    // odczytaj komórkę o adresie first+sizeof(int)
    // przesun marker
    i = va_arg( marker, int);
    // odczytaj komórkę o adresie marker+sizeof(double)
    // przesun marker
    d = va_arg( marker, double ) ;
    // zeruj marker
    va_end( marker );
}
```

foo(7, 3, 2.5)

2.5	ESP+16
	ESP+12
3	ESP+8
7 (first)	ESP+4
Adres powrotu	ESP

Funkcje fread i fwrite

Funkcje te służą do zapisu i odczytu pojedynczych elementów lub tablic. Elementy zapisywane są bez modyfikacji, stąd funkcje wymagają ustawienia binarnego trybu dostępu do pliku.

Deklaracje

```
size_t fwrite( const void *buffer, size_t size,  
              size_t count, FILE *stream );
```

```
size_t fread( void *buffer, size_t size,  
             size_t count, FILE *stream );
```

buffer

adres obszaru pamięci, gdzie znajdują się dane do zapisu lub gdzie dane mają zostać odczytane

size

rozmiar zapisywanego elementu

count

liczba elementów do zapisu

stream

wskaźnik do struktury opisującej plik

Funkcje fread i fwrite (2)

Typowe wywołanie:

```
TYPE var;
```

```
TYPE table[COUNT];
```

```
fwrite(&var, sizeof(var), 1, fout);
```

```
fwrite(table, sizeof(TYPE), COUNT, fout);
```

```
fread(&var, sizeof(var), 1, fin);
```

```
fread(table, sizeof(TYPE), COUNT, fin);
```

Funkcje `fread` oraz `fwrite` są często używane do zapisu danych przetwarzanych przez program we własnym formacie.

Zazwyczaj używa się struktur danych, dla których pamięć jest przydzielana dynamicznie.

Funkcje fread i fwrite (3)

```
typedef struct vectorOfDouble
{
    int count;
    double*elements;
}VectorOfDouble;

int writeVector(const VectorOfDouble*v,FILE*file) {
    fwrite(&v->count,sizeof(v->count),1,file);
    fwrite(v->elements,sizeof(double),v-> count, file);
    return 1;
}

int readVector(VectorOfDouble*v,FILE*file) {
    fread(&(v->count),sizeof(v->count),1,file);
    v->elements = (double*)malloc(v->count*sizeof(double));
    if(!v->elements)return 0;
    fread(v->elements,sizeof(double),v->count,file);
    return 1;
}
```

Funkcje fread i fwrite (4)

Na ogół format zapisu plików nie jest przenośny. Ten sam kod skompilowany dla różnych platform może produkować różne pliki binarne. Problemem jest:

- Reprezentacja danych (na przykład typ `int` może mieć rozmiar 2 lub 4 bajtów)
- Bity mogą być uporządkowane od lewej do prawej lub odwrotnie
- Bajty mogą być zapisywane w kolejności młodszy-starszy lub odwrotnie
- W przypadku zapisanych struktur na postać pliku binarnego ma wpływ wyrównanie i upakowanie pól.

Zabezpieczenia:

- Definiując format ustalamy, że określone elementy formatu mają określoną wielkość (np.: 1, 2, 4 bajty) i rzutujemy typy standardowe na typy zapisywane.
- Zamiast całych struktur zapisujemy ich indywidualne pola

Funkcja `fseek` i `ftell` (1)

- Dostęp do plików specjalnych (`stdin`, `stdout`, plików urządzeń) jest dostępem sekwencyjnym. Oznacza to, że zapisujemy lub odczytujemy strumień danych.
- W przypadku plików dyskowych możliwy jest dostęp swobodny (*random-access*). Możemy przejść do dowolnego miejsca pliku i tam dokonać zapisu lub odczytu.
- Ma to zastosowanie, kiedy plik jest duży i jego zawartość nie mieści się w pamięci, a opracowane algorytmy umożliwiają ograniczenie się do poszczególnych obiektów zapisanych w pliku (np.: rekordów bazy danych).
- Struktura `FILE` zawiera adres bieżącego miejsca w otwartym pliku. Operacje odczytu i zapisu przesuwają ten wskaźnik.

Funkcja fseek i ftell (2)

Funkcja `fseek` pozwala na przesuwanie wskaźnika.

```
int fseek( FILE *stream, long offset, int origin );
```

`origin`

`SEEK_CUR` – bieżące położenie

`SEEK_END` – koniec pliku

`SEEK_SET` – początek pliku

`offset`

dodatnia lub ujemna wielkość przesunięcia

Funkcja `long ftell(FILE*)` podaje bieżącą pozycję wskaźnika pliku.

Funkcje niebuforowanego dostępu do pliku

Funkcje zdefiniowane w bibliotece `<stdio.h>` realizują buforowany dostęp do pliku. Oznacza to, że z każdym otwartym plikiem związany jest bufor o rozmiarze powyżej kilkuset bajtów.

- Operacje zapisu stopniowo wypełniają bufor. W momencie, kiedy jest on pełny bufor jest zapisywany do pliku.
- Podczas operacji odczytu bufor jest wstępnie wypełniany i kolejne dane są odczytywane z bufora. W momencie, kiedy bufor jest pusty kolejna porcja danych jest odczytywana z dysku.

Funkcje niebuforowanego dostępu do plików zdefiniowane są w `<io.h>`. Posługują się one deskryptorem pliku – całkowitoliczbowym indeksem w systemowej tablicy plików otwartych przez dany proces (program).

<code>open</code>	otwiera plik, zwraca całkowitoliczbowy deskryptor (handle)
<code>_sopen</code>	otwiera plik ustawiając flagi współbieżnego dostępu (zakaz/zezwozenie na pisanie lub czytanie)
<code>close</code>	zamyka plik
<code>read</code>	odczytuje dane i umieszcza w buforze
<code>write</code>	zapisuje dane z bufora
<code>lseek</code>	przesuwa wskaźnik bieżącego położenia

Funkcje niebuforowanego dostępu do pliku

- Funkcje buforowane są realizowane za pośrednictwem funkcji niebuforowanych.
- Za pośrednictwem funkcji niebuforowanych implementowane są także strumienie w C++
- Systemowe deskryptory plików umożliwiają dostęp do dodatkowych informacji o pliku (atrybutów, daty i czasu)