Programowanie imperatywne

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

e-mail: pszwed@agh.edu.pl

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

Aktualizacja: 16.05.2020

10. 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 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

Bibliteka *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*.

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

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.

`filename`	Nazwa pliku. Może zawierać pełną ścieżkę, np: `"c:\\mojepliki\\plik.txt"`
`mode`	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.

Przykład – wyszukiwanie tekstu

```
void shiftLeft(char*buf,int size)
{
        int i;
        for(i=0;i<size-1;i++)buf[i]=buf[i+1];
}


void addCharacter(char*buf,int c)
{
        int idx;
        idx=strlen(buf);
        buf[idx]=c;
        buf[idx+1]=0;
}
```

Przykład – wyszukiwanie tekstu

```
void search(const char *pattern)
{
    int c;
    char buf[256]="";
    size_t patternSize=0;

    patternSize=strlen(pattern);
    for(;;){
        c=getchar();
        if(c==EOF)break;
        putchar(c);

        addCharacter(buf,c);
        if(strlen(buf)>patternSize)shiftLeft(buf,strlen(buf)+1);

        if(patternSize!=0 && !stricmp(buf,pattern)){
                printf("<FOUND text=\'%s\'/>", buf);
        }
    }
}
```

Przykład – wyszukiwanie tekstu

```
void replace(const char*pattern,const char*repl)
{
        int c;
        char buf[256]="";
        size_t patternSize=0;
        patternSize=strlen(pattern);

        for(;;){
                c=getchar();
                if(c==EOF)break;
                addCharacter(buf,c);
                if(strlen(buf)>patternSize){
                        putchar(buf[0]);
                        shiftLeft(buf,strlen(buf)+1);
                }
                if(patternSize!=0 && !stricmp(buf,pattern)){
                        printf("%s", repl);
                        buf[0]=0;
                }
        }
        printf("%s", buf);
}
```

Przykład – wyszukiwanie tekstu

```
int main(int argc, const char**argv)
{
    if(argc<2){
        fprintf(stderr,"Call %s search_text  [replace_text]\n",
                                            argv[0]);
        return -1;
    }
    if(argc==2){
        search(argv[1]);
    }

    if(argc>=3){
        if(argc>3){
            fprintf(stderr,"Argument %s ignored\n",argv[3]);
        }
        replace(argv[1],argv[2]);
    }

    return 0;
}
```

Wywołania

```
>search-replace.exe Ala
Ala ma kota
Ala<FOUND text='Ala'/> ma kota
^Z
```

```
>search-replace.exe Ala ma
Ala ma kota
ma ma ko^Z
ta
```

```
>search-replace.exe "Ala ma"
Ala ma kota
Ala ma<FOUND text='Ala ma'/> kota
^Z
```

CTRL-Z ENTER

Kończy strumień wejściowy

getchar() zwraca EOF

Wywołania

```
>search-replace.exe addCharacter dodajZnak < main.c > main2.c
```

```c
void dodajZnak(char*buf,int c)
{
        int idx;
        idx=strlen(buf);
        buf[idx]=c;
        buf[idx+1]=0;
}

void search(const char *pattern)
{
        int c;
        char buf[256]="";
        size_t patternSize=0;

        patternSize=strlen(pattern);
        for(;;){
                c=getchar();
                if(c==EOF)break;
                putchar(c);
                dodajZnak(buf,c);
                if(strlen(buf)>patternSize)shiftLeft(buf,strlen(buf)+1);
                if(patternSize!=0 && !stricmp(buf,pattern)){
                        printf("<FOUND text=\'%s\'/>", buf);
                }
        }
```

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 )
{
   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);
}
```

```
Hello world
12
```

Funkcje o zmiennej liczbie argumentów (3)

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

Przykład

```
void myprint(const char*fmt, ...){
    char buf[64];
    int ival;
    double dval;
    const char*sval;

    va_list marker;
    va_start( marker, fmt );

    const char*last = fmt;
    for(char*ptr=strchr(fmt,'%');ptr;ptr=strchr(ptr+1,'%')){
        switch(*(ptr+1)){
            case 'c':
                ival = va_arg( marker, int);
                buf[0]=ival;
                buf[1]=0;
                break;
            case 'd':
                ival = va_arg( marker, int);
                itoa(ival,buf,10);
                break;
```

Przykład

```
// kontynuacja…

            case 'b':
                ival = va_arg( marker, int);
                itoa(ival,buf,2);
                break;
            case 'o':
                ival = va_arg( marker, int);
                itoa(ival,buf,8);
                break;
            case 'x':
                ival = va_arg( marker, int);
                itoa(ival,buf,16);
                break;
            case 'f':
                dval = va_arg( marker, double);
                sprintf(buf,"%f",dval); //
                break;
```

Przykład

```
// kontynuacja …
            case 's':
                 sval = va_arg( marker, const char*);
                 strcpy(buf,sval);
                 break;
         }
         while(last<ptr)putchar(*(last++));
         last=ptr+2;
         for(char*p=buf;*p;p++)putchar(*p);
    }
    va_end( marker );
    while(*last)putchar(*(last++));
}
```

```
int main(){
    int i = 12345;
    myprint("i=%d (bin:%b, oct:%o hex:%x)\nf=%f\ns=\"%s\"\n",
            i,i,i,i,1.2345,"ala ma kota");

}
```

```
i=12345 (bin:11000000111001, oct:30071 hex:3039)
f=1.234500
s="ala ma kota"
```

Problem – w GCC zniknęło itoa

```
char* itoa(int v,char*buf,int base){
    int i=0;
    if(v<0){
        if(base!=2){
            buf[0]='-';
            v=-v;
        }else {
            buf[0]='1';
            v=v+INT_MAX+1;
        }
        buf++;
    }
    do{
        int digit = v%base;
        buf[i++]=digit<10 ? digit+'0':digit-10+'a';
        v/=base;
    }while(v>0);
    buf[i]=0;
    int n = strlen(buf);
    for(i=0;i<n/2;i++){
        char tmp = buf[n-i-1];
        buf[n-i-1]=buf[i];
        buf[i]=tmp;
    }
    return buf;
}
```

Własna implementacja - powinna zadziałać?

Pliki tekstowe

Zapisujemy do pliku tablicę liczb całkowitych w trybie tekstowym

```
int save_text_mode(const char*name,int*tab,int n){
   FILE*f=fopen(name,"wt");
   if(!f)return 0;
   for(int i=0;i<n;i++)fprintf(f,"%d ",tab[i]);
   fclose(f);
   return 1;

}
```

Plik zawiera tekstową reprezentację liczb (kody ASCII cyfr).

```
933 743 262 529 700 508 752 256 256 119 711 351 843 705
108 393 330 366 169 932 917 847 972 868 980 223 549 592
164 169 551 427 190 624 635 920 944 310 862 484 363 301
710 236 876 431 929 397 675 491 190 344 134 425 629 30
727 126 743 334 104 760 749 620 256 932 572 613 490 509
…
678 708 855 67 273 225 401 426 565 287 299 724 916 949
```

Pliki tekstowe

Odczyt z pliku. Funkcja `fscanf()` zwraca wartość ≤ 0 w przypadku błędu (koniec strumienia). Wskaźnik `ptab` i `n` to wskaźniki do zmiennych, które otrzymają informacje o przydzielonej pamięci i długości tablicy.

```
int load_text_mode(const char*name,int**ptab,int*n){
    FILE*f=fopen(name,"rt");
    if(!f)return 0;
    *ptab=0;
    *n=0;
    for(;;){
        int v;
        if(fscanf(f,"%d",&v)<=0)break;
        *ptab=realloc(*ptab,(*n+1)*sizeof(int));
        (*ptab)[*n]=v;
        (*n)++;
    }
    fclose(f);
    return 1;

}
```

Pliki tekstowe

```
int main(){

    // zapis
    int size = 1000;
    int*tab = malloc(size*sizeof(int));
    for(int i=0;i<size;i++)tab[i]=rand()%size;
    save_text_mode("dane.txt",tab,size);
    free(tab);

    //odczyt
    load_text_mode("dane.txt",&tab,&size);
    printf("size = %d\n",size);
    for(int i=0;i<size;i++){
        printf("%d: %d\n",i,tab[i]);
    }
    free(tab);

}
```

```
size = 1000
0: 933
1: 743
2: 262
3: 529
4: 700
5: 508
6: 752
7: 256
…
990: 273
991: 225
992: 401
993: 426
994: 565
995: 287
996: 299
997: 724
998: 916
999: 949
```

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.

Pliki binarne

```
int save_bin_mode(const char*name,int*tab,int n){
    FILE*f=fopen(name,"wb");
    if(!f)return 0;
    fwrite(tab,sizeof(int),n,f);
    fclose(f);
    return 1;

}
```

Tak wygląda zapis tekstowy – kody ASCII cyfr (933 to szesnastkowo 39,33,33)

```
0000000000: 39 33 33 20 37 34 33 20 | 32 36 32 20 35 32 39 20  933 743 262 529
0000000010: 37 30 30 20 35 30 38 20 | 37 35 32 20 32 35 36 20  700 508 752 256
0000000020: 32 35 36 20 31 31 39 20 | 37 31 31 20 33 35 31 20  256 119 711 351
0000000030: 38 34 33 20 37 30 35 20 | 31 30 38 20 33 39 33 20  843 705 108 393
0000000040: 33 33 30 20 33 36 36 20 | 31 36 39 20 39 33 32 20  330 366 169 932
```

Tak wygląda zapis binarny – 4-bajtowe liczby (933 to 000003A5)

```
0000000000: A5 03 00 00 E7 02 00 00 | 06 01 00 00 11 02 00 00  Ą♥  ç☻  ♠☺  ◄☻
0000000010: BC 02 00 00 FC 01 00 00 | F0 02 00 00 00 01 00 00  Ľ☻  ü☺  đ☻   ☺
0000000020: 00 01 00 00 77 00 00 00 | C7 02 00 00 5F 01 00 00   ☺  w   Ç☻  _☺
0000000030: 4B 03 00 00 C1 02 00 00 | 6C 00 00 00 89 01 00 00  K♥  Á☻  l   ‰☺
0000000040: 4A 01 00 00 6E 01 00 00 | A9 00 00 00 A4 03 00 00  J☺  n☺  ©   ¤♥
```

Pliki binarne

```
int load_bin_mode(const char*name,int**ptab,int*n){
    FILE*f=fopen(name,"rb");
    if(!f)return 0;
    *ptab=0;
    *n=0;
    for(;;){
        int v;
        if(fread(&v,sizeof(int),1,f)==0)break;
        *ptab=realloc(*ptab,(*n+1)*sizeof(int));
        (*ptab)[*n]=v;
        (*n)++;
    }
    fclose(f);
    return 1;
}
```

- Funkcja `fread()` zwraca liczbę przeczytanych bajtów. Jeżeli zwróci 0, to oznacza, że napotkano koniec pliku.
- Wynik wywołania w `main()` taki sam.

Funkcje fread i fwrite (3)

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

Przykład – struktura Vector

```
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;
}
```

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.

Jak odczytać długość pliku?

```
long file_length(const char*name){
    FILE*f = fopen(name,"rb");
    if(!f)return -1;
    fseek(f,0,SEEK_END);
    long s=ftell(f);
    fclose(f);
    return s;
}

int main(){
    long s = file_length("w-pustyni.txt");
    printf("%ld\n",s);
}
```

Przesuwamy kursor na koniec i odczytujemy jego pozycję

643327

Ładujemy plik tekstowy do pamięci

```
char* load_file(const char*name){
    FILE*f = fopen(name,"rb");
    if(!f)return 0;
    fseek(f,0,SEEK_END);
    long s=ftell(f);
    fseek(f,0,SEEK_SET);
    char*txt = malloc((size_t)s+1);
    fread(txt,sizeof(char),(size_t)s,f);
    fclose(f);
    txt[(size_t)s]=0;
    return txt;
}

int main(){
    char*txt = load_file("w-pustyni-utf.txt");
    for(int i=0;i<256;i++)putchar(txt[i]);

    free(txt);
}
```

Henryk Sienkiewicz

W pustyni i w puszczy

ROZDZIAŁ I

— Wiesz, Nel — mówił Staś Tarkowski do swojej przyjaciółki, małej Angielki — wczoraj przyszli zabtie (policjanci) i aresztowali żonę dozorcy Smaina i jej troje dzieci — t

Wracamy do dawnych czasów…

- Zaimplementujemy algorytm sortowania *heapsort* działający na pliku, a nie na tablicy w pamięci. Źródło: *T. H. Cormen , C. E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms, MIT Press, 2009*
- Zakładamy, że plik jest otwarty w trybie pozwalającym na zapis i odczyt `"w+b"` i zapisaliśmy do niego pewną liczbę danych typu `double`.

```
int main(){
    FILE*f=fopen("table.dat","w+b");
    if(!f){
        printf("Blad otwarcia ");
        exit(-1);
    }
    int n = 100;
    for(int i=0;i<n;i++){
        double v = rand()%100/10.0;
        fwrite(&v,sizeof(double),1,f);
    }
    print_file_double(f);
    heapsort(f, n);
    print_file_double(f);
    fclose(f);
}
```

Argumentem funkcji **`heapsort()`** będzie wskaźnik `FILE*` (strumień), a nie tablica.

Podczas sortowania wielokrotnie będzie przesuwany kursor pliku i odczytywane oraz zapisywane elementy…

Heapsort

```
void swap(FILE*f, int a, int b){
    double va,vb;
    a*= sizeof(double);
    b*= sizeof(double);

    fseek(f,a,SEEK_SET);
    fread(&va,sizeof(double),1,f);
    fseek(f,b,SEEK_SET);
    fread(&vb,sizeof(double),1,f);

    fseek(f,b,SEEK_SET);
    fwrite(&va,sizeof(double),1,f);
    fseek(f,a,SEEK_SET);
    fwrite(&vb,sizeof(double),1,f);

}
```

Funkcja zamienia miejscami elementy o indeksach a i b

Heapsort

```
int greater_than(FILE*f,int a, int b){
    double va,vb;
    a*= sizeof(double);
    b*= sizeof(double);

    fseek(f,a,SEEK_SET);
    fread(&va,sizeof(double),1,f);

    fseek(f,b,SEEK_SET);
    fread(&vb,sizeof(double),1,f);
    return va > vb;

}
```

Funkcja porównuje elementy o indeksach a i b.

Heapsort

```
int left(int i){
    return 2*(i+1)-1;
}
int right(int i){
    return 2*(i+1);
}

void max_heapify(FILE*f, int heapsize, int i){
    int l=left(i);
    int r=right(i);
    int largest=i;
    if(l<heapsize && greater_than(f,l,i)){
        largest=l;
    }
    if(r<heapsize && greater_than(f,r,largest)){
        largest=r;
    }
    if(largest!=i){
        swap(f,i, largest);
        max_heapify(f, heapsize, largest);
    }
}
```

Implementacja `max_heapify()` działająca na pliku.

Operator porównania zastąpiony wywołaniem funkcji `greater_then()`.

Heapsort

```
void build_heap(FILE*f, int size){
    for(int i=(size)/2-1; i>=0; i--){
        max_heapify(f, size, i);
    }
}

void heapsort(FILE*f, int size){
    build_heap(f, size);
    printf("~~~\n");
    for(int i=size-1; i>0; i--){
        swap(f,0, i);
        max_heapify(f, i, 0);
    }

}
```

Implementacje funkcji `build_heap()` oraz `heapsort()`, w których tablica została zastąpiona strumieniem.

Heapsort

Funkcja do wydruku zawartości pliku

```
void print_file_double(FILE*f){
    fseek(f,0,SEEK_SET);
    for(;;){
        double v;
        if(fread(&v,sizeof(double),1,f)==0)break;
        printf("%f ",v);
    }
    printf("\n");

}
```

```
int main(){
    FILE*f=fopen("table.dat","w+b");
    // dla n=10
…
    print_file_double(f);
    heapsort(f, n);
    print_file_double(f);
    fclose(f);
}
```

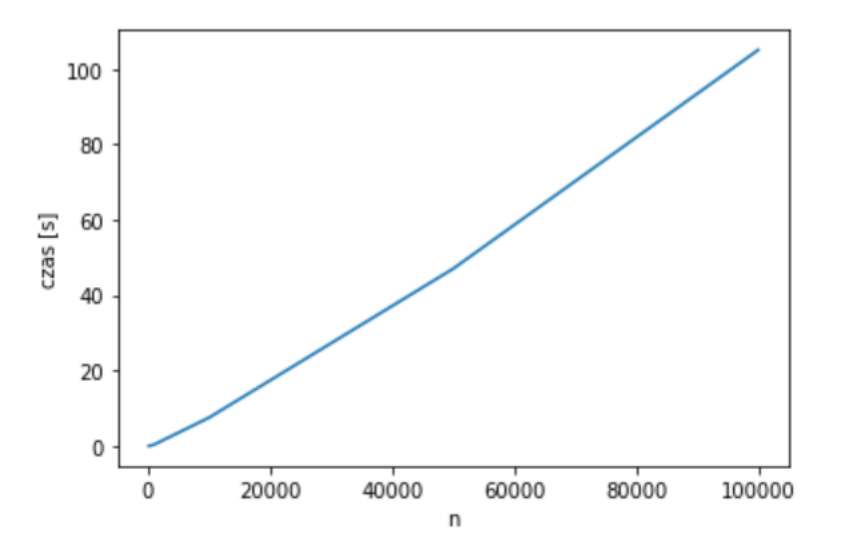
```
3.300000 4.300000 6.200000 2.900000
0.000000 0.800000 5.200000 5.600000
5.600000 1.900000
~~~
0.000000 0.800000 1.900000 2.900000
3.300000 4.300000 5.200000 5.600000
5.600000 6.200000
```

Heapsort

Złożoność obliczeniowa heapsort to $O(n \cdot \log_2 n)$. Czyli czas wykonania $t(n) \leq C \cdot n \cdot \log_2 n$, dla $n > N$. Stała C jest znacznie większa w przypadku pliku, niż w przypadku pamięci.

Wyniki pomiaru czasu:

- n=100 time: 0.031000s
- n=500 time: 0.187000s
- n=1000 time: 0.421000s
- n=10000 time: 7.561000s
- n=50000 time: 47.016000s
- n=100000 time: 105.015000s



Dla n=100000 stała C ma wartość $6.32 \cdot 10^{-5}$

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).

`open`	otwiera plik, zwraca całkowitoliczbowy deskryptor (handle)
`_sopen`	otwiera plik ustawiając flagi współbieżnego dostępu (zakaz/zezwolenie na pisanie lub czytanie)
`close`	zamyka plik
`read`	odczytuje dane i umieszcza w buforze
`write`	zapisuje dane z bufora
`lseek`	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)