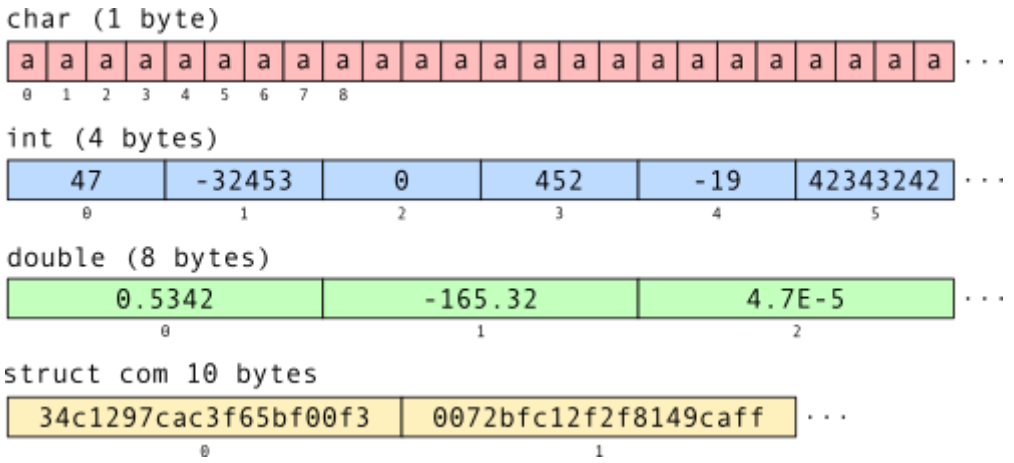
Arquivos binários

Video desta aula

Todos os arquivos contêm sequências de bytes, mas costuma-se dizer que um arquivo é “binário” quando seu conteúdo não é uma informação textual (ou seja, não representa um texto usando codificações como ASCII, UTF-8 ou outras). Arquivos binários são usados para armazenar informações mais complexas como imagens, música, código executável, etc.

Em C, um arquivo binário é visto como uma sequência de blocos de mesmo tamanho. O tamanho dos blocos depende do tipo de informação armazenada no arquivo. Por exemplo, um arquivo de números reais `double` terá blocos de 8 bytes, enquanto um arquivo de caracteres (`char`) terá blocos de 1 byte, como mostra a figura:



Lembre-se que o SO só armazena a sequência de bytes, sem considerar nem registrar o tamanho dos blocos. **Cabe à aplicação** definir o tamanho de bloco que deseja usar em cada arquivo.

Leitura/escrita de blocos

A linguagem C oferece funções para ler e escrever blocos de bytes em arquivos, que efetuam a cópia desses bytes da memória para o arquivo ou vice-versa.

A funções a seguir permitem ler/escrever blocos de bytes em arquivos binários. Todas essas funções estão definidas no arquivo `stdio.h`.

Lê até `count` blocos de tamanho `size` bytes cada um e os deposita no vetor `data`, a partir do *stream* indicado. Retorna o número de blocos lidos:

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

Escreve até `count` blocos de tamanho `size` bytes do vetor `data` no *stream* indicado. Retorna o número de blocos escritos:

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

Essas funções também podem ser usadas para ler/escrever em arquivos-texto, pois textos são sequências de blocos de 1 byte.

Exemplo de uso

Este exemplo manipula um arquivo binário `numeros.dat` contendo números reais. São implementadas (em arquivos separados) as operações de escrita de números no arquivo, listagem e ordenação do conteúdo:

escreve.c

```c
// Escreve N valores reais aleatórios em um arquivo, em formato binário

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

#define ARQUIVO "numeros.dat"
#define NUMVAL 10

int main (int argc, char *argv[])
{
  FILE* arq ;
  int i, ret ;
  float value[NUMVAL] ;

  // abre o arquivo em modo "append"
  arq = fopen (ARQUIVO, "a") ;
  if (!arq)
  {
    perror ("Erro ao abrir arquivo") ;
    exit (1) ;
  }

  // inicia gerador de números aleatórios
  srand (clock()) ;

  // gera NUMVAL valores aleatórios reais
  for (i = 0; i < NUMVAL; i++)
    value[i] = random() / 100000.0 ;

  // escreve os valores gerados no final do arquivo
  ret = fwrite (value, sizeof (float), NUMVAL, arq) ;
  if (ret)
    printf ("Gravou %d valores com sucesso!\n", ret) ;
  else
    printf ("Erro ao gravar...\n") ;

  // fecha o arquivo
  fclose (arq) ;
  return (0) ;
}
```

lista.c

```c
// Lista o conteúdo de um arquivo que contém números reais em formato binário

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

#define ARQUIVO "numeros.dat"

int main (int argc, char *argv[])
{
  FILE* arq ;
  float value ;

  // abre o arquivo em modo leitura
  arq = fopen (ARQUIVO, "r") ;
  if (!arq)
  {
    perror ("Erro ao abrir arquivo") ;
    exit (1) ;
  }

  // lê e imprime os valores contidos no arquivo
  fread (&value, sizeof (float), 1, arq) ;
  while (! feof (arq))
  {
    printf ("%f\n", value) ;
    fread (&value, sizeof (float), 1, arq) ;
  }

  // fecha o arquivo
  fclose (arq) ;
  return (0) ;
}
```

ordena.c

```c
// Lê os números reais de um arquivo binário, os ordena e os escreve de volta
no arquivo

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

#define ARQUIVO "numeros.dat"
#define MAXVAL 100000

float value[MAXVAL] ;
int num_values ;

int main (int argc, char *argv[])
{
  FILE* arq ;
  int i, j, menor ;
  float aux ;
```

```
  // abre o arquivo em leitura/escrita, preservando o conteúdo
  arq = fopen (ARQUIVO, "r+") ;
  if (!arq)
  {
    perror ("Erro ao abrir arquivo") ;
    exit (1) ;
  }

  // lê números do arquivo no vetor
  num_values = fread (value, sizeof (float), MAXVAL, arq) ;
  printf ("Encontrei %d números no arquivo\n", num_values) ;

  // ordena os números (por seleção)
  for (i = 0; i < num_values-1; i++)
  {
    // encontra o menor elemento no restante do vetor
    menor = i ;
    for (j = i+1; j < num_values; j++)
      if (value[j] < value[menor])
        menor = j ;

    // se existe menor != i, os troca entre si
    if (menor != i)
    {
      aux = value[i] ;
      value[i] = value[menor] ;
      value[menor] = aux ;
    }
  }

  // retorna o ponteiro ao início do arquivo
  rewind (arq) ;

  // escreve números do vetor no arquivo
  fwrite (value, sizeof (float), num_values, arq) ;

  // fecha o arquivo
  fclose (arq) ;
  return (0) ;
}
```

No arquivo `ordena.c`, o conteúdo inteiro do arquivo é lido com **apenas uma** chamada `fread` e escrito com apenas uma chamada `fwrite`. Isso é perfeitamente possível, desde que a estrutura usada para receber os dados na memória coincida byte a byte com a forma como eles estão dispostos no arquivo.

Posicionamento no arquivo

Para cada arquivo aberto em uma aplicação, o sistema operacional mantém um contador interno indicando a posição da próxima operação de leitura ou escrita. Esse contador é conhecido como **ponteiro de posição** (embora não seja realmente um ponteiro).

Por default, as operações em um arquivo em C ocorrem em posições sucessivas dentro do arquivo: cada leitura (ou escrita) corre **após** a leitura (ou escrita) precedente, até atingir o final do arquivo. Essa forma de acesso ao arquivo é chamada de **acesso sequencial**.

Por vezes uma aplicação precisa ler ou escrever em posições específicas de um arquivo, ou precisa voltar a ler uma posição do arquivo que já percorreu anteriormente. Isso ocorre frequentemente em aplicações que manipulam arquivos muito grandes, como vídeos ou bases de dados. Para isso é necessária uma forma de **acesso direto** a posições específicas do arquivo.

Em C, o acesso direto a posições específicas de um arquivo é feita através de funções de **posicionamento de ponteiro**, que permitem alterar o valor do ponteiro de posição do arquivo, antes da operação de leitura/escrita desejada.

Todas as funções de manipulação do ponteiro de arquivo consideram as **posições em bytes** a partir do início do arquivo, nunca em número de blocos.

As funções mais usuais para acessar o ponteiro de posição de um arquivo em C são indicadas a seguir.

Ajusta posição do ponteiro no *stream* indicado:

```
int fseek (FILE* stream, long int offset, int whence)
```

O ajuste é definido por `offset`, enquanto o valor de `whence` indica se o ajuste é relativo ao início do arquivo (SEEK_SET), à posição corrente (SEEK_CUR) ou ao final do arquivo (SEEK_END). Ver também `fseeko` e `fseeko64`. Exemplos:

```
                              // posiciona o ponteiro de "arq":
fseek (arq, 1000, SEEK_SET) ; // 1000 bytes após o início
fseek (arq, -300, SEEK_END) ; // 300 bytes antes do fim
fseek (arq, -500, SEEK_CUR) ; // 500 bytes antes da posição atual
```

Reposiciona o ponteiro no início (posição 0) do *stream* indicado:

```
void rewind (FILE* stream)
```

Informa a posição corrente de leitura/escrita em `stream` (ver também `ftello` e `ftello64` no manual).

```
long int ftell (FILE* stream)
```

Outras funções

Para truncar (“encurtar”) um arquivo, deixando somente os primeiros `length` bytes:

```
#include <unistd.h>
#include <sys/types.h>

// usando o nome do arquivo, sem abri-lo
int truncate (const char *path, off_t length);

// usando um descritor UNIX (fd)
int ftruncate (int fd, off_t length);
```

Para obter as propriedades (metadados) de um arquivo:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

// usando o nome do arquivo, sem abri-lo
int stat  (const char *pathname, struct stat *statbuf);

// usando um descritor UNIX (fd)
int fstat (int fd, struct stat *statbuf);
```

As informações sobre o arquivo serão depositadas pelo núcleo na estrutura `statbuf`, cujo conteúdo está descrito abaixo. Os campos da estrutura são detalhados na página de manual da função `fstat()`.

```
struct stat
{
  dev_t     st_dev;         /* ID of device containing file */
  ino_t     st_ino;         /* Inode number */
  mode_t    st_mode;        /* File type and mode */
  nlink_t   st_nlink;       /* Number of hard links */
  uid_t     st_uid;         /* User ID of owner */
  gid_t     st_gid;         /* Group ID of owner */
  dev_t     st_rdev;        /* Device ID (if special file) */
  off_t     st_size;        /* Total size, in bytes */
  blksize_t st_blksize;     /* Block size for filesystem I/O */
  blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */
  struct timespec st_atim;  /* Time of last access */
  struct timespec st_mtim;  /* Time of last modification */
  struct timespec st_ctim;  /* Time of last status change */
  #define st_atime st_atim.tv_sec  /* For backward compatibility */
  #define st_mtime st_mtim.tv_sec
  #define st_ctime st_ctim.tv_sec
};
```

Exercícios

1. Escreva três programas C separados para:
    1. escrever um arquivo com n (n é um número aleatório > 100) inteiros `long` aleatórios, armazenados em modo binário;
    2. ler o arquivo de inteiros em um vetor, ordenar o vetor e reescrever o arquivo;
    3. escrever na tela os primeiros 10 números e os últimos 10 números contidos no arquivo.
    4. Utilize stat ou fstat para recuperar o tamanho do arquivo

Mais exercícios no capítulo 11 da apostila do NCE/UFRJ.