

# GBC053 - Gerenciamento de Bancos de Dados

## Aula 4 Conceitos Fundamentais de Estruturas de Arquivos

Humberto Razente  
humberto.razente@ufu.br

# Arquivo como *stream*

- Suponha que deseja-se construir um arquivo com nomes e endereços. Algoritmo:

open OUTPUT file

get LAST name as input

while ( LAST name has length > 0)

    get FIRST name, ADDRESS, CITY, STATE, ZIP as input

    write LAST, FIRST to file OUTPUT

    write ADDRESS to file OUTPUT

    write CITY, STATE, ZIP to file OUTPUT

    get LAST name as input

endwhile

close OUTPUT

# Arquivo como *stream*

- Se os seguintes nomes e endereços são inseridos:

John Ames  
123 Maple  
Stillwater, OK 74075

Alan Mason  
90 Eastgate  
Ada, OK 74820

a saída para o arquivo conterá:

```
AmesJohn123 MapleStillwaterOK74075MasonAlan90 EastgateAdaOK74820
```

# Arquivo como *stream*

- O programa escreveu a informação com precisão, como especificado
  - um *stream* de bytes, sem informações adicionais
  - mas perdeu-se a noção de atributo
- Problema
  - como separar os atributos?
    - "John Ames" ou "123 Maple"

# Arquivo como *stream*

- Atributo: noção conceitual
  - necessidade: organizar os arquivos de modo que atributos possam ser recuperados
- Métodos convencionais:
  - 1) forçar que atributos tenham tamanho definido
  - 2) iniciar cada atributo com um indicador de tamanho
  - 3) colocar um delimitador no final de cada atributo para separá-lo do próximo
  - 4) usar expressões tipo "chave=valor"

# Arquivo como *stream*

- 1) Forçar que atributos tenham tamanho definido
  - pode-se definir um *struct* com os atributos
  - simplifica escrita da estrutura no arquivo e sua leitura
  - linguagem C dispõe da função ***sizeof***
  - ***fread*** e ***fwrite*** preparados para ler/escrever ***structs***
  - uma matemática simples é suficiente para recuperar dados em termos dos atributos originais
  - desvantagem: armazena espaços não utilizados

# Arquivo como *stream*

- Em C:

```
struct {  
    char last[10];  
    char first[10];  
    char address[15];  
    char city[15];  
    char state[2];  
    char zip[9];  
} set_of_fields;
```

# Arquivo como *stream*

- 2) Iniciar cada atributo com um indicador de tamanho

```
4Ames4John10123 Mapple10Stillwater  
2OK5740755Mason4Alan1190 Eastgate3  
Ada2OK574820
```

- se atributos não tiverem mais que 256 caracteres, pode-se armazenar tamanho em 1 byte



# Arquivo como *stream*

- 3) Colocar um delimitador no final de cada atributo para separá-lo do próximo
  - escolha de um caracter especial
  - caracter escolhido não poderá ser empregado como entrada de dados
    - barra vertical (pipe): "|"
    - espaço: " "
    - ...

```
Ames|John|123 Mapple|Stillwater|  
OK|74075|Mason|Alan|90 Eastgate|  
Ada|OK|74820|
```

# Arquivo como *stream*

- 4) Usar expressões tipo "chave=valor"
  - dados vem com informação da estrutura
  - bom formato para atributos faltantes
  - desvantagem: espaço ocupado pela estrutura

```
LAST=Ames | FIRST=John | ADDRESS=123  
Maple | CITY=Stillwater | STATE=OK |  
ZIP=74075 | LAST=Mason | FIRST=Alan |  
ADDRESS=90 Eastgate | CITY=Ada |  
STATE=OK | ZIP=74820 |
```

# Estrutura de registros

- Registro
  - conjunto de atributos que pertencem a um mesmo elemento quando o arquivo é visto em termos de uma organização de alto nível
  - é uma noção conceitual
  - nível de organização imposto aos dados para preservar um significado

# Estrutura de registros

- **Métodos empregados para organizar um arquivo de registros:**
  - requerer que registros tenham um número de bytes pré-definido
  - requerer que registros tenham um número de atributos pré-definido
  - iniciar cada registro com um indicador de tamanho consistindo da contagem do número de bytes do registro
  - usar um segundo arquivo para armazenar o endereço do byte de início de cada registro
  - colocar um delimitador no final de cada registro para separá-lo do próximo

# Estrutura de registros

- Requerer que registros tenham um número de bytes pré-definido

Ames	Mary	123 Maple	Stillwater	OK74075
Mason	Alan	90 Eastgate	Ada	OK74820

(a)

Ames	Mary	123 Maple	Stillwater	OK	74075	← Unused space →
Mason	Alan	90 Eastgate	Ada	OK	74820	← Unused space →

(b)

# Estrutura de registros

- Requerer que registros tenham um número de atributos pré-definido
  - ao invés de especificar número de bytes, especifica-se número de atributos
  - no arquivo, a cada  $m$  atributos, tem-se um registro

```
Ames | Mary | 123 | Maple | Stillwater | OK | 74075 | Mason | Alan | 90 | Eastgate | Ada | OK | . . .
```

(c)

# Estrutura de registros

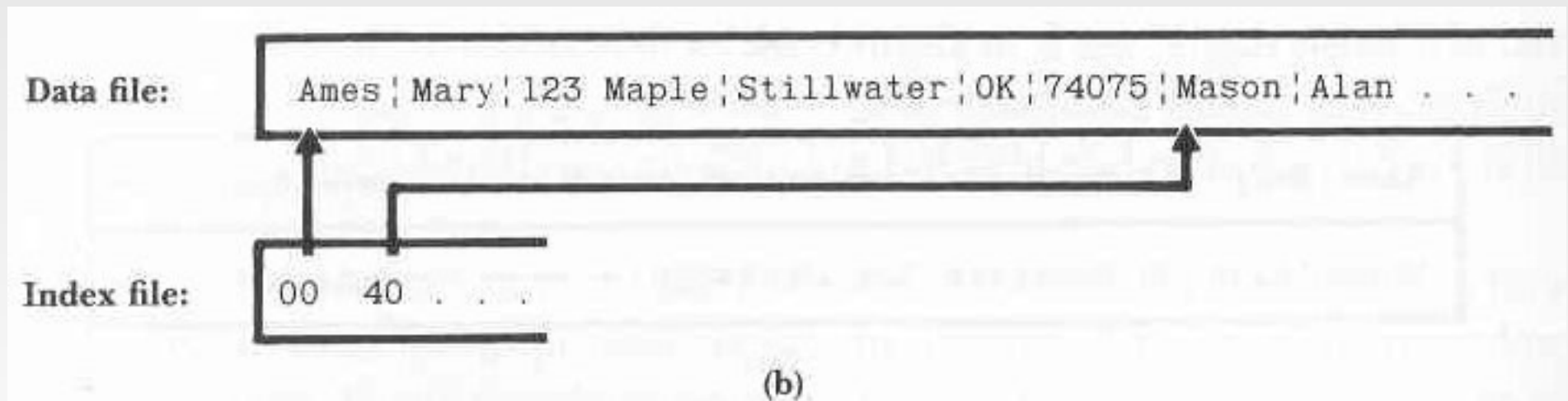
- Iniciar cada registro com um indicador de tamanho consistindo da contagem do número de bytes do registro
  - comum quando precisa-se armazenar registros de tamanho variável

```
40Ames|Mary|123 Maple|Stillwater|OK|74075|36Mason|Alan|90 Eastgate . . .
```

(a)

# Estrutura de registros

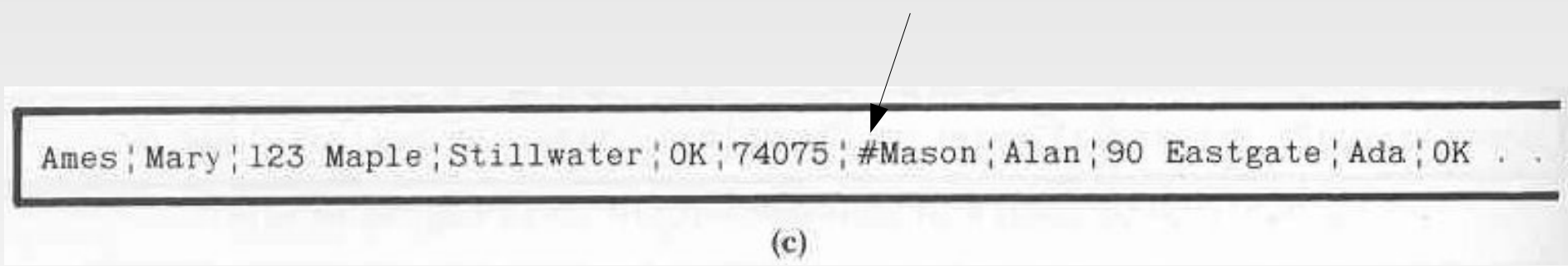
- Usar um segundo arquivo para armazenar o endereço do byte de início de cada registro
  - arquivo de índice





# Estrutura de registros

- Colocar um delimitador no final de cada registro para separá-lo do próximo



Ames, Mary, 123 Maple, Stillwater, OK, 74075, #Mason, Alan, 90 Eastgate, Ada, OK . . .

The diagram shows a horizontal rectangular box containing a text string. An arrow points from above to the '#' character in the string, which serves as a record delimiter. The text string is: Ames, Mary, 123 Maple, Stillwater, OK, 74075, #Mason, Alan, 90 Eastgate, Ada, OK . . .

(c)

# Acesso a registros

- ao procurar por um registro, é conveniente identificá-lo com uma **chave** baseada no conteúdo do registro
  - em geral é necessário que chaves sejam distintas
  - caso não sejam distintas, é possível usar chaves secundárias

# Acesso a registros

- Acesso sequencial
  - ler um arquivo, registro após registro, até encontrar um registro ou atingir o final do arquivo
  - nas próximas aulas, o acesso sequencial será empregado como base para comparação com métodos mais elaborados
    - medidas de comparação: número de acessos a disco (READ)
  - em geral, acesso sequencial é caro para a maioria das aplicações

# Acesso a registros

- Acesso direto
  - seek para uma determinada posição
  - custo de  $O(1)$ , enquanto acesso sequencial tem custo de  $O(n)$
  - para realizar acesso direto, é preciso conhecer a posição do início do registro
    - arquivo de índices

# Arquivo de registros

- Se objetivo é armazenar registros de tamanho fixo em arquivo, pode-se tentar encaixar registros em um setor
  - por exemplo, com tamanho 512 ou outra potência de 2
  - de modo que um registro não fique dividido em dois setores

# Registros de cabeçalho

- muitas vezes é necessário armazenar informações gerais do arquivo para auxiliar no uso futuro do arquivo
  - um registro de cabeçalho é geralmente colocado no início do arquivo
    - como número de registros armazenados
    - comprimento dos registros
    - informações sobre versão, etc
  - o registro de cabeçalho normalmente tem conteúdo diferente dos registros do arquivo

# Modelos abstratos de dados

- Nos primórdios da computação, computadores eram usados para processar registros
  - armazenados em fita
  - código era escrito em cartões perfurados
- Gradualmente, os usuários passaram a reconhecer que computadores poderiam ser usados para mais que atributos e registros
  - processamento e transmissão de áudio e vídeo
  - imagens e documentos
  - etc

# Modelos abstratos de dados

- É mais simples pensar em documentos, imagens, vídeos como **objetos** ao invés de uma sequência de atributos e registros
- Essa noção está relacionada com o modelo abstrato de dados
  - visão dos dados orientada pela aplicação ao invés de visão dos dados orientada pela mídia de armazenamento
  - permite guardar no próprio objeto informações que o software pode usar para entender o formato, tirando do usuário essa responsabilidade
    - exemplo: arquivos ".jpg", ".avi", ".wav"



# Modelos abstratos de dados

- Quando um *header* de arquivo contém informações sobre o formato, pode-se dizer que o arquivo é *self-describing*
- Suponha por exemplo que são armazenados as seguintes informações em um arquivo:
  - o nome de cada atributo
  - o número de bytes de cada atributo
  - o número de atributos por registro
- É possível escrever um programa que leia e imprima um arquivo com qualquer número de atributos por registro?

# Modelos abstratos de dados

- Metadados
  - são dados que descrevem os dados primários em um arquivo
  - podem ser incorporados em qualquer arquivo cujos dados primários requerem informação de suporte
  - exemplo: precisa-se desenvolver um tipo de dados para armazenar imagens de um satélite
    - informações sobre a latitude e longitude que cada pixel corresponde é informação de suporte para o dado primário, ou seja, a imagem

# Portabilidade e padronização

- O compartilhamento de arquivos significa que um arquivo possa ser aberto/reproduzido em diferentes computadores / sistemas operacionais / arquiteturas

# Portabilidade e padronização

- Diferenças entre sistemas operacionais
  - Exemplo: quebra de linha
    - Linux/Unix: `"\n"`
    - Windows / DOS: `"\r\n"`

# Portabilidade e padronização

- Diferenças na arquitetura dos computadores
  - fwrite de uma variável int com valor 32 ( $32_{10} = 20_{16}$ ) em um programa em C em um computador Sun 3
    - hex: 0020
  - mesmo programa, em um micro padrão IBM PC:
    - hex: 2000
- Em ambos os casos os valores foram escritos da maneira que estão representados na memória RAM
  - a Sun armazena primeiro o byte de mais alta ordem
  - o PC armazena primeiro o byte de mais baixa ordem

# Portabilidade e padronização

- Conversão entre tabelas de caracteres
  - EBCDIC
  - ASCII
  - Unicode
- Conversão de tipos numéricos
  - entre plataformas
    - pode-se optar por converter os valores para um tipo padrão, i.e. IEEE
    - conversão de ponto flutuante pode acarretar em perda de precisão

# Exercício 1 – Implementação

- Criar um programa em C ou C++ que leia o arquivo contendo CEPs no formato

```
69903620\tAC\tRIO BRANCO\tTRAVESSA RAMOS FERREIRA\n
```

- e escreva em outro arquivo no formato

```
TRAVESSA RAMOS FERREIRA|RIO BRANCO|AC|69903620\n
```

- Escreva uma função para busca no arquivo

# Exercício 2 – Implementação

- Altere o programa do exercício 1 para ler os dados para um arranjo da seguinte estrutura:

```
struct item {  
    int cep;  
    char uf[3];  
    char cidade[39];  
    char logradouro[67];  
} item;
```

- e escreva essa estrutura em outro arquivo, um registro seguido do outro a cada ***sizeof(item)***
- Escreva uma função para busca no arquivo



# Exercício 3 – Implementação

- Altere o programa do exercício 1 para gerar um arquivo de índice contendo a posição inicial de cada registro
- O programa deve permitir o acesso direto a registros por meio da posição do registro
  - exemplo: se o usuário solicitar o registro de posição 1000, o programa deve ler no índice a posição inicial do milésimo registro e fazer um **seek** para ler o registro

# Leitura complementar

- Capítulo Fundamental File Structures Concepts do livro
  - Folk et al. "File Structures: An Object-Oriented Approach with C++", Editora Pearson, 3ª edição, 1998