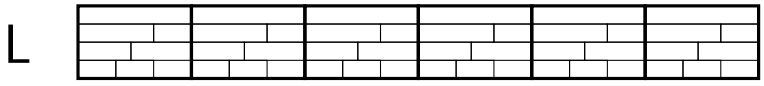
Fundamentos de programação

aula 09: Programando bancos de dados

1 Introdução

Em sua versão mais simples, um banco de dados é só uma lista onde cada posição armazena várias informações diferentes



Por exemplo, o banco de dados de uma clínica para animais poderia conter as seguintes informações

- **tipo**: `gato, cachorro, passarinho, peixe, cobra`
- **nome, dono**
- **peso, idade**
- **tem pulgas:** `sim, não, não se aplica`

e por aí vai.

Para criar essa lista, o primeiro passo é criar um *registro* que contém essas informações.

E a linguagem C oferece dois recursos úteis para trabalhar com registros.

O primeiro deles é a noção de *tipo enumerado*, que serve para definir categorias de valores.

Por exemplo, o tipo do animal é uma categoria que pode ser definida da seguinte maneira

```
enum especie = { gato, cachorro, passarinho, peixe, cobra };
```

Daí, se nós declaramos a variável

```
enum especie X;
```

então nós podemos escrever no nosso programa

```
X = gato;
```

Quer dizer, agora a linguagem C reconhece `gato`, `cachorro`, etc. como valores que podem ser atribuídos a variáveis do tipo `enum especie`.

O segundo recurso é a noção de `struct`, que permite definir o registro propriamente dito.

Abaixo nós temos a definição dos registros para o banco de dados da clínica de animais

```
enum especie  =  { gato, cachorro, passarinho, peixe, cobra };
enum opcao    =  { sim, nao, nao_se_aplica };
```

```
struct animal
 {
   char          nome[50], dono[50];
   int           anos, meses;
   float         peso;
   enum especie  tipo;
   enum opcao    tem_pulgas;
 };
```

Quer dizer, o nosso registro é apenas uma coleção de variáveis, agrupadas sob o nome `struct animal`.

Daí, se nós declararmos a variável

```
struct animal X;
```

nós podemos acessar os campos individualmente da seguinte maneira

```
X.anos = 2;   X.meses = 8

if ( X.tem_pulgas == sim )  printf("%s tem pulgas", X.nome);
```

e por aí vai.

O nosso primeiro exemplo com registros é um programa que lê as informações de um animal do teclado, e depois imprime essas informações na tela

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

enum especie {gato, cachorro, passarinho, peixe, cobra};
enum opcao {sim, nao, nao_se_aplica};

struct animal
 {
   char          nome[50], dono[50];
   int           anos, meses;
   float         peso;
   enum especie  tipo;
   enum opcao    tem_pulgas;
 };
struct animal X;

int main()
  {
     system("clear");

     printf("Informações sobre um novo animal\n");

     printf("Nome: ");    fgets(X.nome, 50, stdin);
     printf("\nDono: ");  fgets(X.dono, 50, stdin);

     printf("\nTipo: (1) gato  (2) cachorro  (3) passarinho  (4) peixe  (5) cobra\n\n");
     printf("Escolha uma opção: ");
     scanf("%d", &X.tipo);

     printf("\nIdade: ");  scanf("%f", &X.idade);
     printf("\nPeso: ");   scanf("%f", &X.peso);
```

```
    printf("\nTem pulgas: (1) sim  (2) não  (3) não se aplica \n\n");
    printf("Escolha uma opção: ");
    scanf("%d", &X.tem_pulgas);

    printf("\n\n\nRelatório:\n");

    printf("Nome:  %s\n", X.nome);
    printf("Dono:  %s\n", X.dono);
    printf("Tipo:  %d\n", X.tipo);
    printf("Idade: %2d e %2d meses\n", X.anos, X.meses);
    printf("Peso:  %2.2f\n\n", X.peso);
    printf("Tem pulgas:  %d\n", X.tem_pulgas);
    printf("\n\n");
}
```

Observações:

- A instrução

  ```
  system("clear");
  ```

  apenas limpa a tela para iniciar a execução do programa.

- A instrução

  ```
  fgets(X.nome, 50, stdin);
  ```

  lê uma sequência de caracteres do teclado, e armazena na variável indicada como primeiro argumento (`X.nome`, nesse caso).

  Infelizmente, a versão mais simples

  ```
  gets(X.nome);
  ```

  não é segura, pois ela pode ler (e escrever na memória) uma quantidade de caracteres maior do que aquela alocada para a variável `X.nome`.

  Nós também poderíamos usar

  ```
  scanf("%s", X.nome);
  ```

  mas essa instrução interrompe a leitura no primeiro espaço em branco.

- A instrução

  ```
  scanf("%d", &X.tipo);
  ```

  lê um número inteiro do teclado e armazem essa informação em `X.tipo`
  — mas é preciso não esquecer o símbolo `&`.

  Quer dizer, o símbolo & é uma indicação de que aquele é o lugar onde você quer guardar a informação — (apenas uma pequena esquisitice da linguagem C).

- A instrução

  ```
  scanf("%f", &X.idade);
  ```

  lê um `float` (i.e., número decimal com precisão simples) do teclado, e armazena essa informação em `X.idade`.

2 Consultando o banco de dados

Agora imagine que o banco de dados já contém informações sobre vários animais, e nós queremos realizar consultas.

Por exemplo, imagine que nós queremos ver os registros de todos os gatos de 3 anos.

Então, nós poderíamos escrever um trecho de programa assim

```
for (i=0; i<M; i++)
 {
   if ( B[i].tipo == gato && B[i].anos == 3 )
    {
     printf("Nome:  %s\n", B[i].nome);
     printf("Dono:  %s\n", B[i].dono);
     printf("Tipo:  %d\n", B[i].tipo);
     printf("Idade: %2d e %2d meses\n", B[i].anos, B[i].meses);
     printf("Peso:  %2.2f\n\n", B[i].peso);
     printf("Tem pulgas:  %d\n", B[i].tem_pulgas);
     printf("\n\n");
    }
```

Agora suponha que nós queremos consultar todos os cachorros que tem pulgas.

Então o trecho de programa seria esse aqui

```
for (i=0; i<M; i++)
 {
   if ( B[i].tipo == cachorro && B[i].tem_pulgas == sim )
    {
     printf("Nome:  %s\n", B[i].nome);
     printf("Dono:  %s\n", B[i].dono);
     printf("Tipo:  %d\n", B[i].tipo);
     printf("Idade: %2d e %2d meses\n", B[i].anos, B[i].meses);
     printf("Peso:  %2.2f\n\n", B[i].peso);
     printf("Tem pulgas:  %d\n", B[i].tem_pulgas);
     printf("\n\n");
    }
```

É basicamente a mesma coisa, não é?

Quer dizer, tudo o que mudou foram os campos que estão sendo testados, e os valores desses campos.

Daí que, nós poderíamos escrever um trecho de programa que corresponde a uma consulta genérica, e utilizá-lo para realizar todas as consultas desse tipo.

Para fazer isso, nós definimos um registro auxiliar

```
struct animal X;
```

e colocamos nesse registro os valores da consulta.

Daí, basta percorrer o banco de dados, e imprimir aqueles registros que possuem os mesmos valores dos campos (não vazios) de `X`.

O programa abaixo implementa essa ideia.

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

#define N 10

enum especie {nao_importa, gato, cachorro, passarinho, peixe, cobra};
enum opcao   {n_importa, sim, nao, nao_se_aplica};

struct animal
 {
   char nome[50];
   char dono[50];
   enum especie  tipo;
   float  idade, peso;
   enum opcao tem_pulgas;
 };
struct animal B[N] = { {"juquinha", "Sr Silva", gato, 3, 2.3, sim} ,
                       {"valadao", "Bianca", passarinho, 1, 0.5, sim},
                       {"biju", "Aninha", gato, 3, 3.2, nao} ,
                       {"dengosa", "Sra Silva", cobra, 45, 18.0, nao_se_aplica},
                       {"sansao", "Sr Luis", gato, 2, 3.2, nao} ,
                       {"alexander", "Sr Silva", peixe, 1, 0.2, nao_se_aplica} ,
                     };
struct animal X;
int  i, M = 6;

int main()
  {
     system("clear");
     printf("Consulta ao banco de dados de animais\n");

     printf("Nome: ");    fgets(X.nome, 50, stdin);
     printf("\nDono: ");  fgets(X.dono, 50, stdin);

     printf("\nTipo: (1) gato  (2) cachorro  (3) passarinho  (4) peixe  (5) cobra\n\n");
     printf("Escolha uma opção (0 p/ não importa): ");
     scanf("%d", &X.tipo);

     printf("\nIdade (0 p/ não importa): ");  scanf("%f", &X.idade);
     printf("\nPeso  (0 p/ não importa): ");  scanf("%f", &X.peso);

     printf("\nTem pulgas: (1) sim  (2) não  (3) não se aplica \n\n");
     printf("Escolha uma opção (0 p/ não importa): ");
     scanf("%d", &X.tem_pulgas);

     system("clear");
     printf("Resultado da consulta\n");

     for (i=0; i<M; i++)
      {
        if ( ( strncmp(B[i].nome,X.nome,strlen(B[i].nome)) == 0 ||
               strcmp(X.nome,"\n") == 0 ) &&
             ( strncmp(B[i].dono,X.dono,strlen(B[i].dono)) == 0 ||
               strcmp(X.dono,"\n") == 0 )  &&
             ( B[i].tipo  == X.tipo  || X.tipo  == nao_importa )  &&
             ( B[i].peso  == X.peso  || X.peso  == 0  )  &&
             ( B[i].idade == X.idade || X.idade == 0  )  &&
             ( B[i].tem_pulgas == X.tem_pulgas || X.tem_pulgas == n_importa )
           )
```

```
          {
           printf("Nome:  %-10s %*c", B[i].nome, 4, ' ');
           printf("Dono:  %-10s\n", B[i].dono);
           printf("Tipo:  %d\n", B[i].tipo);
           printf("Idade: %2.2f %*c", B[i].idade, 10, ' ');
           printf("Peso:  %2.2f\n", B[i].peso);
           printf("Tem pulgas:  %d\n", B[i].tem_pulgas);
           printf("\n\n");
  }   }  }
```

2.1 Separando a consulta da impressão

Certo.

O nosso programa já funciona.

Mas, nós podemos deixar as coisas mais organizadas separando a etapa da consulta da etapa da impressão dos resultados.

E a pequena esperteza que permite fazer isso consiste em definir uma lista que armazena o resultado da consulta

```
int C[N];     // resultados da consulta
int nC;       // número de registros recuperados
```

Note que isso é uma lista de inteiros.

E a ideia é que a lista `C` armazena os índices das posições do banco de dados que satisfazem os critérios da consulta.

Daí, uma vez que a gente tem esses índices, basta imprimir os registros que estão nessas posições.

Pronto.

Agora o programa pode ser reorganizado em 3 etapas:

```
int main()
  {
     // ETAPA 1: obter os critérios da consulta

     system("clear");
     printf("Consulta ao banco de dados de animais\n");

     ( . . . )


     // ETAPA 2: realização da consulta ao banco de dados

     nC = 0;
     for (i=0; i<M; i++)
      {
        if ( . . . )    // compara registro B[i] com valores em X
         {
           C[nC] = i;  nC++;
         }
```

```
// ETAPA 3: impressão dos resultados da consulta

system("clear");
printf("Resultado da consulta\n");

if ( nC == 0 )
   printf ("Nenhum resultado encontrado\n");
else
   for (j=0; j<nC; j++)
    {
      i = C[j];
      printf("Nome:  %-10s %*c", B[i].nome, 4, ' ');
      ( . . . )
    }
```

3 Consultas mais legais

Agora imagine que nós adicionamos os seguintes campos ao nosso registro de animais

```
char      pai[50], mae[50];
enum sexo  sexo;             // macho ou femea
```

Com base nessas novas informações, nós podemos descobrir, por exemplo, quem são os filhos do gato `Sansão`.

Para isso, basta fazer

```
strcpy(X.pai, "Sansao");

// ETAPA 2: realização da consulta ao banco de dados
( . . . )

// ETAPA 3: impressão dos resultados da consulta
( . . . )
```

Observação:

- Na linguagem C nós não podemos atribuir diretamente um valor a uma variável que armazena texto

  ```
  X.pai  =  "Sansão";       // instrução inválida
  ```

  As 3 alternativas para fazer isso são:

  - inicializar a variável no momento da declaração

    ```
    char  T[50]  =  "Tem dias que a gente se sente ...";
    ```

  - fazer a leitura do teclado

    ```
    fgets(X.dono, 50, stdin);
    ```

  - utilizar a função `strcpy` da biblioteca `string.h` (como foi feito acima)

    ```
    strcpy(X.pai, "Sansao");
    ```

Certo.

Mas nós também podemos querer descobrir quem são os netos de `Sansão`.

Para fazer isso, basta descobrir quem são os filhos dos gatos que apareceram na consulta anterior.

E aqui nós utilizamos a lista auxiliar `C` outra vez.

Quer dizer, depois da primeira consulta, a lista `C` contém as posições do banco de dados `B` onde estão os filhos de `Sansão`.

E daí, basta repetir a consulta para cada um desses gatos.

Mas, para a coisa funcionar direito, é preciso ter alguns cuidados:

1. Quando o filho de `Sansão` é macho, nós devemos fazer a consulta utilizando o campo `pai`

   ```
   strcpy(X.pai, B[i].nome);
   ```

   E quando o filho de `Sansão` é fêmea, nós devemos fazer a consulta utilizando o campo `mae`

   ```
   strcpy(X.mae, B[i].nome);
   ```

   Além disso, como nós vamos estar reutilizando a variável `X` nas diversas consultas, é preciso apagar qualquer valor que possa ter sido colocado lá por uma consulta anterior

   ```
   strcpy(X.pai, B[i].nome);
   strcpy(X.mae, " ");
   ```

2. A ideia é reutilizar o código da `ETAPA 2` sem fazer nenhuma modificação.

   Mas, esse código armazena os resultados da consulta na variável auxiliar `C`.

   Portanto, antes de começar, nós precisamos copiar as posições dos filhos de `Sansão` para outro lugar.

   Para isso, nós declaramos outra variável auxiliar D

   ```
   int  D[N],  nD;
   ```

   Além disso, a medida que os filhos dos filhos de Sansão vão sendo encontrados, eles também precisam ser armazenados em outro lugar.

   Para isso, nós declaramos mais uma variável auxiliar

   ```
   int  E[N],  nE;
   ```

3. Pode acontecer do mesmo gato aparecer em mais de uma consulta.

   Isso acontece quando o gato é filho de um filho e uma filha de *Sansão*

   (*Hmm, isso é muito comum no mundo dos gatos*)

   Mas, nós não queremos apresentar esse gato duas vezes no resultado da nossa consulta.

   Para evitar isso, nós nos certificamos de que a lista `E` (que armazena os filhos dos filhos) não contém elementos repetidos.

   E para fazer isso, nós verificamos se o elemento já está na lista `E` antes de colocá-lo lá.

4. Finalmente, a ideia também é reaproveitar o código da `ETAPA 3` sem fazer nenhuma modificação.

   Mas, esse código imprime os registros das posições armazenadas em `C`.

   Logo, quando nós terminamos de fazer a nossa consulta, nós colocamos o seu resultado em `C` para poder fazer a impressão.

Abaixo nós temos o esqueleto do programa que implementa essas ideias:

```
 . . .

// CONSULTA 1: armazena os filhos de Sansão em C
( . . . )


// Copiar conteúdo de C para D

for ( j=0; j<nC; j++ )   D[j] = C[j];
nD = nC;


// Realizar consultas com os gatos que estão em D

for ( j=0; j<nD; j++ )
 {
    i = D[j];
    if ( B[i].sexo == macho )
     {
       strcpy(X.pai, B[i].nome); strcpy(X.mae, "");
     }
    else
     {
       strcpy(X.pai, ""); strcpy(X.mae, B[i].nome);
     }

    // ETAPA 2: realização da consulta ao banco de dados

    ( . . . )

    // Copiar resultados em C para a lista E (sem duplicatas)

    for ( k=0; k<nC; k++ )
     {
        Aux = 0;
        for ( l=0; l<nE; l++)  if ( E[l] = C[k] )  Aux = 1;

        if ( Aux == 0 )
         {
           E[nE] = C[k];  nE++;
         }
 }   }

// Copiar conteúdo de E para C

for ( j=0; j<nE; j++ )   C[j] = E[j];
nC = nE;

// ETAPA 3: impressão dos resultados em C

( . . . )
```

Outros exemplos:

a. **O irmão mais pesado de `Sansão`**

Agora imagine que nós queremos encontrar o irmão mais pesado de `Sansão`.

Bom, nós consideramos que um irmão de `Sansão` é qualquer gato que tenha o mesmo pai ou a mesma mãe que `Sansão`.

Para fazer essa consulta, primeiro nós temos que encontrar o registro de `Sansão`

```
for (i=0; i<M; i++)
  if ( strncmp (B[i].nome, "Sansão", strlen("Sansão") ) == 0 )
     break;
```

Daí, nós copiamos o nome de seu pai e sua mãe para o registro auxiliar `X`

```
strcpy (X.pai, B[i].pai);
strcpy (X.mae, B[i].mae);
```

A seguir, nós observamos que nós não podemos usar a consulta genérica dessa vez.

Quer dizer, a consulta genérica encontra os registros que tem os mesmos valores que todos os campos não vazios de `X`.

Mas dessa vez, só o pai ou só a mãe bastam.

Portanto, nós vamos reescrever o laço que realiza a consulta novamente.

```
nC = 0;
for (i=0; i<M; i++)
  if ( B[i].tipo == gato  &&
       ( strncmp (B[i].nome, X.pai, strlen(X.pai) ) == 0 ||
         strncmp (B[i].nome, X.mae, strlen(X.mae) ) == 0 )  )
   {
     C[nC] = i;  nC++;
   }
```

Finalmente, basta examinar todos os irmãos de `Sansão` (cujas posições foram armazenadas em `C`) para descobrir quem é o mais pesado

```
if ( nC == 0 )
   printf("Não foi encontrado nenhum irmão de Sansão\n");
else
 {
   i = C[0];  Maior = B[i].peso;   pos = i;

   for (j=1; j<nC; j++)
    {
       i = C[j];
       if ( B[i].peso > Maior )
        {
          Maior = B[i].peso;   pos = i;
    }   }
   printf("O irmão mais pesado de Sansão é %s\n", B[pos].nome);
 }
```

◇

b. **O primo mais pesado de `Sansão`**

Dessa vez, nós queremos encontrar o primo mais pesado de `Sansão`.

Para isso, é preciso encontrar os tios de `Sansão` (i.e., os irmãos de seu pai e de sua mãe).

E para isso é preciso encontrar os avós de `Sansão`.

Nós vamos começar por aqui.

Quer dizer, primeiro nós localizamos o registro de `Sansão`, e obtemos os nomes dos pais dele

```
for (i=0; i<M; i++)
  if ( strncmp (B[i].nome, "Sansão", strlen("Sansão") ) == 0 )
     break;

strcpy (X.pai, B[i].pai);
strcpy (X.mae, B[i].mae);
```

Daí, nós localizamos os registros dos pais, para obter os nomes dos avós

```
for (i=0; i<M; i++)
  if ( strncmp (B[i].nome, X.pai, strlen(X.pai) ) == 0 )
     break;

strcpy (Y.pai, B[i].pai);
strcpy (Y.mae, B[i].mae);

for (i=0; i<M; i++)
  if ( strncmp (B[i].nome, X.mae, strlen(X.mae) ) == 0 )
     break;

strcpy (Z.pai, B[i].pai);
strcpy (Z.mae, B[i].mae);
```

Certo.

Agora nós fazemos uma consulta para descobrir as posições dos tios de `Sansão`

```
nC = 0;
for ( i=0; i<M; i++ )
 {
  if ( B[i].tipo == gato  &&
       ( strncmp (B[i].nome, Y.pai, strlen(Y.pai) ) == 0 ||
         strncmp (B[i].nome, Y.mae, strlen(Y.mae) ) == 0 ||
         strncmp (B[i].nome, Z.pai, strlen(Z.pai) ) == 0 ||
         strncmp (B[i].nome, Z.mae, strlen(Z.mae) ) == 0 )  )
   {
     C[nC] = i;  nC++;
   }
 }
```

Legal.

Nesse ponto nós já temos as posições dos tios de `Sansão` na lista auxiliar `C`.

Daí, os primos de `Sansão` são aqueles que possuem o pai ou a mãe nessa lista
— (*e que não são o próprio* `Sansão`, *claro*)

O trecho de código abaixo encontra esses registros, colocando o resultado na lista auxiliar `D`

```
nC = 0;
for ( i=0; i<M; i++ )
 {
   if ( strncmp (B[i].nome, "Sansao", strlen("Sansão") ) != 0 )
    {
       Aux = 0;
       for (j=0; j<nC; j++)
        {
           k = C[j];
           if ( B[i].tipo == gato  &&
                ( strncmp (B[i].nome, B[k].pai, strlen(B[k].pai) ) == 0 ||
                  strncmp (B[i].nome, B[k].mae, strlen(B[k].mae) ) == 0 )  )
            {
              Aux = 1; break;
       }   }
       if ( Aux == 1 )
        {
           D[nD] = i;  nD++;
 }  }   }
```

Finalmente, basta examinar os primos de `Sansão` para descobrir qual deles é o mais pesado

```
if ( nD == 0 )
   printf("Não foi encontrado nenhum primo de Sansão\n");
else
 {
   i = D[0];  Maior = B[i].peso;   pos = i;

   for (j=1; j<nD; j++)
    {
       i = D[j];
       if ( B[i].peso > Maior )
        {
          Maior = B[i].peso;   pos = i;
    }   }
   printf("O primo mais pesado de Sansão é %s\n", B[pos].nome);
 }
```

◇

c. **Todos os descendentes de `Sansão`**

Relembre que no início da seção nós encontramos os filhos e netos de `Sansão`.

Agora, nós vamos encontrar todos os seus descendentes.

Isso pode ser feito em dois passos:

1. Primeiro, nós criamos uma lista auxiliar para armazenar os descendentes

   ```
   int  F[N],  nF = 0;
   ```

   E daí, logo após a primeira consulta, nós já colocamos os filhos de `Sansão` lá

   ```
   for (j=0; j<nC; j++)  F[j] = C[j];
   nF =  nC;
   ```

2. Depois, nós colocamos o esqueleto de programa acima dentro de um laço de repetição, para encontrar as sucessivas gerações de descendentes de `Sansão`.

   A ideia aqui é que

   - no início, a lista `D` contém uma geração de descendentes (e.g., os filhos)
   - no final, a lista `E` contém a próxima geração de descendentes (e.g., os netos)

   Daí que, basta copiar a lista `E` novamente para `D` e rodar o programa outra vez, para obter a geração que vem a seguir.

   Mas, para a coisa funcionar direito, é preciso ter algum cuidado.

   O problema é que o mesmo gato pode aparecer em duas gerações diferentes.

   Isso acontece, por exemplo, quando o gato é filho de uma filha e um neto de `Sansão`.

   (*Hmm, isso não é incomum no mundo dos gatos*)

   Porque daí, ele vai ser neto e bisneto de Sansão.

   Nesse caso, nós não queremos colocar esse gato duas vezes na lista `D`, para fazer consultas com ele.

   Para evitar isso, nós só copiamos de `E` para `D` aqueles elementos que não estão em `F`.

   Pronto, é só isso.

   A ideia é continuar encontrando gerações sucessivas de `Sansão`, até que uma hora nós obtemos uma geração vazia.

Abaixo nós temos o esqueleto de programa que implementa essa ideia.

```
 . . .

// CONSULTA 1: armazena os filhos de Sansão em C
( . . . )


// Copiar conteúdo de C para D e F

for ( j=0; j<nC; j++ )  {  D[j] = C[j];  F[j] = C[j];  }
nD = nC;  nF = nC;


while ( nD > 0 )
 {

   // Realizar consultas com os gatos que estão em D

   for ( j=0; j<nD; j++ )
    {

       (  .  .  .  )

    }


   // Copiar resultados em E para as listas D e F (sem duplicatas)

   nD = 0;
   for ( k=0; k<nE; k++ )
    {
       Aux = 0;
       for ( l=0; l<nF; l++)  if ( F[l] = E[k] )  Aux = 1;

       if ( Aux == 0 )
        {
          F[nF] = E[k];  nF++;
          D[nD] = E[k];  nD++;
        }
 }


// Copiar conteúdo de F para C

for ( j=0; j<nF; j++ )   C[j] = F[j];
nC = nF;

// ETAPA 3: impressão dos resultados em C

( . . . )
```

◇