Faça a leitura e depois responda o que se pede:

LEITURA:
O tipo "tData" pode ser definido de qualquer uma das formas abaixo:

```
struct data{
      int dia;
      int mes;
      int ano;
};
typedef struct data tData;
```

```
typedef struct data{
      int dia;
      int mes;
      int ano;
} tData;
```

```
typedef struct{
      int dia;
      int mes;
      int ano;
} tData;
```

```
tData x;  /* declara uma variável x do tipo "TipoData" */
```

É fácil atribuir valores aos campos de um registro:

```
x.dia = 31;
x.mes = 8;
x.ano = 1998;
```

É muito comum usar ponteiros que apontam para registros ou estruturas, ou seja, guardam o endereço de uma estrutura. Por exemplo,

```
tData *p;   /* define ponteiro p para um tipo TipoData */
tData  x;
 p = &x;          /* agora p aponta para x */
(*p).dia = 31;   /* mesmo efeito que x.dia = 31 */
(*p).mes = 8;
(*p).ano = 1998;
```

A expressão `p->dia` é o mesmo que `(*p).dia` :

```
p->dia = 31;     /* mesmo efeito que (*p).dia = 31 */
```

Alocação dinâmica de memória

Alocação dinâmica de memória, ou seja, alocação que ocorre durante a execução de um programa.

A função `malloc` aloca um bloco de *bytes* consecutivos na memória do computador e devolve o endereço desse bloco. Exemplo de uso:

```
tData *ptr;
ptr = (tData *) malloc (sizeof(tData));
if (ptr == NULL) {
   exit (EXIT_FAILURE);
}
ptr->dia = 12;
ptr->mes = 3;
ptr->ano = 2007;
printf ("Data: %d - %d - %d\n", ptr->dia, ptr->mes, ptr->ano );
free (ptr);

ptr = NULL;
```

A expressão `sizeof (tData)` retorna o número de *bytes* de um tData.

A função `malloc` devolve um ponteiro "genérico", ou seja, do tipo **`void *`**, para um bloco de *bytes* consecutivos. No exemplo acima, esse ponteiro é convertido em um ponteiro para o tData através do cast especificado antes do malloc.

Se a memória do computador já estiver toda ocupada, `malloc` não consegue alocar mais espaço e devolve `NULL`.

```
ptr = (tData *) malloc (sizeof(tData));
if (ptr == NULL) {
   exit (EXIT_FAILURE);
}
```

A função `exit` encerra a execução do programa. Com o parâmetro EXIT_SUCCESS ou 0, o sistema operacional é informado que o programa terminou com sucesso, já o contrário ocorre quando se usa o parâmetro EXIT_FAILURE.

A função `free` libera a porção de memória alocada por `malloc`. O comando `free (ptr)` avisa o sistema de que o bloco de *bytes* apontado por `ptr` está "livre", ou seja, pode ser reutilizado pela aplicação. Uma próxima chamada de `malloc` poderá inclusive utilizar esse mesmo bloco de *bytes* que foi liberado pelo comando free. É importante que os ponteiros armazenam o endereço da área liberada através do free recebam o valor NULL.

As funções `exit`, `malloc` e `free` estão na biblioteca stdlib.h.

Se fizermos:

```
tData *p;
int i;
p = (tData *) malloc (10*sizeof(tData));
if (p == NULL) {
   exit (EXIT_FAILURE);
}
```

Se p não for NULL, estará apontado para o início de uma área de memória suficiente para armazenar 5 elementos do tipo tData e podemos interagir sobre ele, como fazemos com um vetor. Veja a continuação do código acima:

```
for (i=0; i < 10; i++){
    printf("Digite uma data: ");
    scanf("%d %d %d", &p[i].dia, &p[i].mes, &p[i].ano);
}
```

Podemos criar um vetor de 500 elemento onde cada elemento desse vetor pode armazenar um endereço de um tData. O usuário pode digitar até 500 datas, mas ele deve decidir se deseja continuar digitando ou encerrar. Veja o exemplo:

```
tData * vet[500]; //observe que essa declaração é diferente de
                  //tData vet[500];
tData * pData;
int i;
int resp;

i = 0;
do{

    pData = (tData *) malloc (sizeof(tData)); //cria o espaço para
                                              //armazenar um tData
    if (pData == NULL) {
       exit (EXIT_FAILURE);
    }

    printf("Digite uma data: ");
    scanf("%d %d %d", &pData->dia, &pData->mes, &pData->ano);

    vet[i] = pData;
```

```
        printf("Deseja digita outra data [1-sim, <outro valor>-nao]? ");
        scanf("%d", &resp);

  }while ((resp == 1) && (i<500));
```

PERGUNTA:
Qual a vantagem desse programa em relação a um programa que tivesse o mesmo propósito, mas trabalhasse com vetor de elementos do tipo tData ao invés de usar um vetor de endereços de tData?

IMPLEMENTE O PROGRAMA SEGUINTE:

Dada a estrutura:

```
typedef struct {
  char nome[50];
  float peso;
  float altura;
  int anoNascimento;
}tPessoa;
```

Sem utilizar variáveis globais faça um programa conforme especificado abaixo:

Pergunte ao usuário a quantidade máxima de pessoas e crie através do malloc espaço suficiente para que essas informações sejam digitadas, ou seja, um vetor de tPessoa do tamanho especificado. Nesse exercício não estamos trabalhando com um vetor de ponteiros. O retorno do malloc deve ser atribuído a uma variável chamada **pessoas**. A seguir exiba o seguinte menu:

MENU PRINCIPAL:

1 – Cadastrar pessoa

2 – Listar pessoas

0 – Sair

Seu programa deve encerrar apenas quando o usuário escolher a opção 0.

Sempre que escolher a opção 1, UMA pessoa será cadastrada. Observe que cada vez que essa opção é escolhida, as informações sobre a pessoa devem ser inseridas em um espaço diferente dentro da região alocada com malloc: na primeira vez será em pessoas[0], na segunda vez, em pessoas[1] e, assim, sucessivamente. Não permita cadastrar mais pessoas do que foi informado no início do programa.

Sempre que escolher a opção 2, o programa exibe as informações das pessoas cadastradas.

ALTERAÇÃO NO PROGRAMA ANTERIOR:
Faça uma nova versão do programa anterior. Não pergunte mais a quantidade de pessoas, mas crie um vetor de tamanho 100 onde cada elemento vetor é do tipo tPessoa *, ou seja, pode armazenar o endereço um espaço do "tipo tPessoa". Ao escolher a opção 1, você deve criar através do malloc o espaço para guardar os dados de uma pessoa e atribuir o endereço desse espaço a um elemento do vetor.