Lista de exercícios 8 – Ponteiros e alocação dinâmica

1. Refaça os exercícios da lista 7 (registros) de forma a evitar a cópia dos registros na chamada e no retorno das funções. Não utilize alocação dinâmica, pois não será necessário. Em geral, as seguintes modificações devem ser feitas:
    a. Para funções que recebem um registro, elas passarão a receber um ponteiro para um registro. Por exemplo, o parâmetro `struct Retangulo ret` se torna `struct Retangulo *ret`
    b. Como o parâmetro passará a ser um ponteiro, ao invés de usar .membro para acessar o membro de um registro, use ->membro.
    c. Para chamar uma função, passe o endereço de um registro ao invés do registro em si. Por exemplo, `area(&retangulo)`.
    d. Para funções que retornam um registro, cuidado para não retornar um ponteiro para um registro declarado dentro da função. A melhor abordagem é mudar a função para não retornar nenhum valor (void) e, dependendo da situação, utilizar uma das seguintes abordagens:
        i. As funções `translada` (exercício 1) e `adiciona` (exercício 3) modificam o registro que foi passado como parâmetro da função.
        ii. As funções `soma` (exercício 2), `novo_conjunto` (exercício 3), `união` (exercício 3), `intersecção` (exercício 3) e `recebe_X` (exercício 4) passam a receber um ponteiro para um registro como parâmetro adicional no qual o resultado será inserido.

2. Na aula sobre registros foi feito um exercício para processar uma compra de supermercado. Foram definidos os seguintes registros:

```c
#define MAX_PRODUTOS 100

struct Produto {
    char nome[50];
    int quantidade;
    float preco;
};

struct Compra {
    struct Produto produtos[MAX_PRODUTOS];
    int n_produtos;           // Número de produtos na compra
    float valor_total;        // Valor total da compra
    char forma_pagamento[50];
};
```

Fizemos então uma função chamada `processa_compra` que recebia do usuário os produtos comprados e retornava uma `struct Compra` contendo os dados. Reescreva a função utilizando alocação dinâmica, o registro `Produto` acima e o seguinte registro `Compra`:

```c
struct Compra {
```

```
    struct Produto *produtos;
    int n_produtos;           // Número de produtos na compra
    float valor_total;        // Valor total da compra
    char forma_pagamento[50];
};
```

Assim como fizemos em aula com a função `recebe_valores`, você pode fazer três versões desta função:

```
/* Retorna uma cópia do registro criado dentro da função */
struct Compra processa_compra(void);

/* Recebe um registro Compra alocado e insere os valores */
void processa_compra(struct Compra *compra)

/* Aloca um registro Compra e retorna um ponteiro para ele */
struct Compra *processa_compra(void)
```

Note que em todas as versões sempre será necessário alocar espaço para o ponteiro `produtos` dentro da função.

3. Em alguns exercícios das aulas sobre registros e ponteiros foi utilizado o seguinte registro para representar uma lista de valores:

```
struct Lista {
    int valores[N_MAX];
    int n;          // Número de valores na lista
    int n_max;      // Tamanho máximo da lista
};
```

Nos exercícios, implementamos as funções `list`, `append` e `pop` para criar uma lista e adicionar e remover elementos. Reescreva essas funções utilizando a seguinte estrutura

```
// Tamanho inicial da lista
#define N_ALLOC_INICIAL 8

struct Lista {
    int *p_valores;
    int n;          // Número de valores na lista
    int n_alloc;    // Tamanho reservado para a lista
};
```

- A função `list` cria uma nova lista na qual apenas 8 valores são alocados para o ponteiro p_valores.
- Na função `append`, quando o 9º valor for adicionado, a memória reservada para p_valores é alocada para o dobro do tamanho (16), e o valor é adicionado. Quando o 17º valor for adicionado, a quantidade de memória de p_valores é dobrada novamente e assim por

diante. Em outras palavras, sempre que não houver mais posições para adicionar um valor na lista, o espaço alocado para ela é dobrado e o valor é adicionado. Dica: Veja o exercício de realocação de vetor feito em aula.

- Na função pop, sempre que o número de elementos da lista se tornar igual à metade do tamanho alocado, a memória é realocada para metade do tamanho.

* Listas do Python são implementadas em C exatamente dessa forma!

Exemplo de uso das funções:

```
struct Lista lista = list();

// Adiciona 8 valores à lista
for (int i = 0; i < 8; i++) {
    append(&lista, i);
}
// Situação atual da lista:
// lista.n: 8, lista.n_alloc: 8

append(&lista, 8);
// Situação atual da lista:
// lista.n: 9, lista.n_alloc: 16

// Adiciona mais 8 valores à lista (17 valores no total)
for (int i = 9; i < 17; i++) {
    append(&lista, i);
}
// Situação atual da lista:
// lista.n: 17, lista.n_alloc: 32

pop(&lista);
// Situação atual da lista:
// lista.n: 16, lista.n_alloc: 16
```