1. Implemente um pequeno formatador de texto em C++ que recebe uma string correspondente a um paragráfo de entrada e retorne uma string com o mesmo texto, mas formatado com texto justificado (ocupando todo o espaço de uma linha de tamanho especificado).

   A justificativa do texto é feita quebrando o texto em linhas menores que o tamanho especificado e inserindo os espaços necessários entre palavras para que o texto ocupe toda a linha. Por simplicidade, assuma que o texto de entrada não possui quebras de linha.

   A função deve ter a assinatura abaixo. O parâmetro "input" corresponde ao texto de entrada; "targetLength" corresponde ao tamanho final da linha que será ocupada pelo texto. É obvio que é possível que sejam definidas quantas funções auxiliares quanto for necessário.

   ```
   std::string justifyParagraph(const std::string& input, int targetLength)
   ```

   Como examplo, dado o texto (o caracter " " está sendo mostrado como "-" para melhor visualização, mas a função deve usar " " mesmo):

   ```
   New-component-JYScrollPaneMap-which-is-pretty-useful----for-large-scrollpane-views
   ```

   com um "targetLength" de 35 deve produzir

   ```
   New----component---JYScrollPaneMap\nwhich--is--pretty-useful-for-arge\nscrollpane-views
   ```

2. Implemente uma versão adicional `justifyDocument(const std::string& input, int targetLength)` também em C++ para que seja possível receber uma entrada de texto contendo múltiplos parágrafos. Lembre-se que parágrafos são separados por quebras de linha na entrada.

3. Especifique como o problema acima poderia ser implementado usando classes. Como exemplo, assuma que a classe principal seria chamada Document, que é formado por uma relação de parágrafos. As classes devem possui um método render() que é responsável por mostrar a string resultante que pode ser apresentada na saída de texto:

   ```
   class Document {
   public:
      Document(const std::string& input);

      std::string render() const;

   private:
      // ...
      std::vector<Paragraph> m_paragraphs;
   };
   ```

```
class Paragraph {
public:
   Paragraph(const std::string& input);

   std::string render() const;

private:
    // ...
};
```

Que mudanças seriam necessárias nas definições para suportar paragráfos com outros tipos de formatação, como por exemplo, centralizado, alinhado à direita ou a esquerda?
Quais as vantagens e desvantagens de implementar essa funcionalidade usando ifs/switches vs herança? Explique.

4. Em geral, quais as vantagens e desvantagens de se usar referencias (&) vs pointers (*) em C++? O que são smart pointers e quais as suas vantagens e desvantagens?

5. Qual o impacto no uso de polimorfismo da definição `std::vector<Paragraph>` versus `std::vector<Paragraph *>` versus `std::vector<std::shared_ptr<Paragraph>>`?

6. Sugira que modificações podem ser feitas no fragmento de código de pseudo C++ abaixo para que seja usado RAII (por meio dos construtores e destrutores) para fechar automaticamente o arquivo aberto sem o uso de try/catch (no exemplo abaixo, File é um tipo fictício com operações open(), close() e read()). Mostre como ficaria a pseudo definição do tipo File.

```
//...
File file;
try {
     file.open("nome do arquivo.txt");

     // Uso do arquivo onde é possível disparar exceção.
     // ...

     file.close();

} catch ( ... ) {
    file.close();

     // Propaga a exceção.
     throw;
}
```