UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS
INE - DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

RELATÓRIO PROJETO I DE ESTRUTURAS DE DADOS

Erik Orsolin de Paula (22102195)

Vitor Praxedes Calegari (22200379)

FLORIANÓPOLIS
2023

1 - Introdução

Neste Projeto I de Estruturas de Dados, somos apresentados a dois problemas práticos relacionados ao mundo da robótica e processamento de dados.

O primeiro desafio é: ler e interpretar arquivos XML que descrevem o ambiente de um robô aspirador. Pede-se exclusivamente a verificação de aninhamento e fechamento das marcações (*tags*) no arquivo XML utilizando o conceito de pilha(LIFO). O segundo desafio nos leva um passo adiante. Com os dados do cenário em mãos, precisamos calcular a área que o robô aspirador deve limpar utilizando o conceito de fila(FIFO). Isso envolve entender a posição inicial do robô e como ele se move pelo espaço.

Ambos os problemas nos dão a chance de aplicar conceitos de estruturas de dados em situações do mundo real, mostrando a utilidade prática do que aprendemos em sala de aula.

2 - Códigos desenvolvidos

main.cpp - parte referente ao exercício 1

```cpp
/// @brief Essa função lê um arquivo dado, guarda as
/// informações em um vetor e retorna o ponteiro desse vetor
/// @param xmlfilename Nome do arquivo que queremos transformar
/// em vetor de caracteres
/// @return Retorna um ponteiro para o buffer alocado em memória
char * arquivo_para_vetor(char * xmlfilename) {

    // Abre o arquivo xml
    ifstream myfile;
    myfile.open(xmlfilename);
    // Pega o ponteiro do buffer
    filebuf* pbuf = myfile.rdbuf();

    // Pega o tamanho do buffer
    size_t size = pbuf->pubseekoff (0,myfile.end,myfile.in);
    pbuf->pubseekpos (0,myfile.in);

    // Aloca memória para os caracteres
    char* buffer=new char[size];

    // Le os caracteres
    pbuf->sgetn (buffer,size);

    // Fecha o arquivo
    myfile.close();

    return buffer;
}
```

A função `arquivo_para_vetor` é responsável por ler um arquivo XML e converter seu conteúdo em um vetor de caracteres, que é então retornado para o chamador.

Detalhamento:

1. **Entrada:**
    - A função aceita um único argumento, `xmlfilename`, que é o nome do arquivo XML que desejamos processar.
2. **Processo de Abertura do Arquivo:**
    - A função começa abrindo o arquivo especificado usando a classe `ifstream`.

3. **Aquisição do Tamanho do Arquivo:**
    - Uma vez que o arquivo está aberto, a função determina o tamanho do arquivo em bytes. Isso é feito usando o método `pubseekoff` para mover o ponteiro de leitura para o final do arquivo e, em seguida, obtendo a posição atual. Isso efetivamente nos dá o tamanho total do arquivo.
4. **Alocação de Memória:**
    - Com o tamanho do arquivo em mãos, a função aloca um buffer de memória suficientemente grande para armazenar todo o conteúdo do arquivo.
5. **Leitura do Arquivo:**
    - O conteúdo do arquivo é então lido diretamente para o buffer de memória usando o método `sgetn`.
6. **Finalização:**
    - Após a leitura, o arquivo é fechado e o ponteiro para o buffer de memória (que contém o conteúdo do arquivo) é retornado.

main.cpp - parte referente ao exercício 1

```
/// @brief Verifica se um dado vetor de caracteres segue a regra
/// de aninhamento de arquivos .xml
/// @param buffer Vetor de caracteres contendo os dados que
/// deseja-se verificar
bool verifica_vetor(char * buffer) {
    structures::ArrayStack<string> pilha(1000);

    char char_atual = buffer[0];
    int tamanho_buffer = string(buffer).size();
    bool resultado = true;

    int i = 0;
    int count = 0;
    while (i < tamanho_buffer) {

        // Entra no laço de uma das chaves do XML
        if (char_atual == '<') {
            count++;
            string temp = string();
            // Le o nome compledo da chave
            while (char_atual != '>' && i < tamanho_buffer) {
                temp.push_back(char_atual);
                char_atual = buffer[++i];
            }
            if (char_atual == 26) {
                break;
            }
            temp.push_back('>');

```

```
            // Checa se é uma chave de abertura ou fechamento

            // Se for de abertura coloca na pilha
            if (temp[1] != '/') {
                pilha.push(temp);
            } else {
                // Se for de fechamento verifica se o topo da pilha
                // tem a chave correspondente, caso não tenha imprime
                // "erro"
                temp.erase(1, 1);
                if (pilha.top() != temp) {
                    resultado = false;
                    break;
                } else {
                    pilha.pop();
                }
            }

        }
```

```
            // Le o próximo caractere
            char_atual = buffer[++i];
        }

        // Se a pilha não estiver vazia ou se não houver nenhuma chave
        if (!pilha.empty() || !count) {
            resultado = false;
        }

        return resultado;
    }
```

A função `verifica_vetor` é projetada para validar se um vetor de caracteres, que representa conteúdo de um arquivo XML, segue as regras de aninhamento corretas para as tags XML.

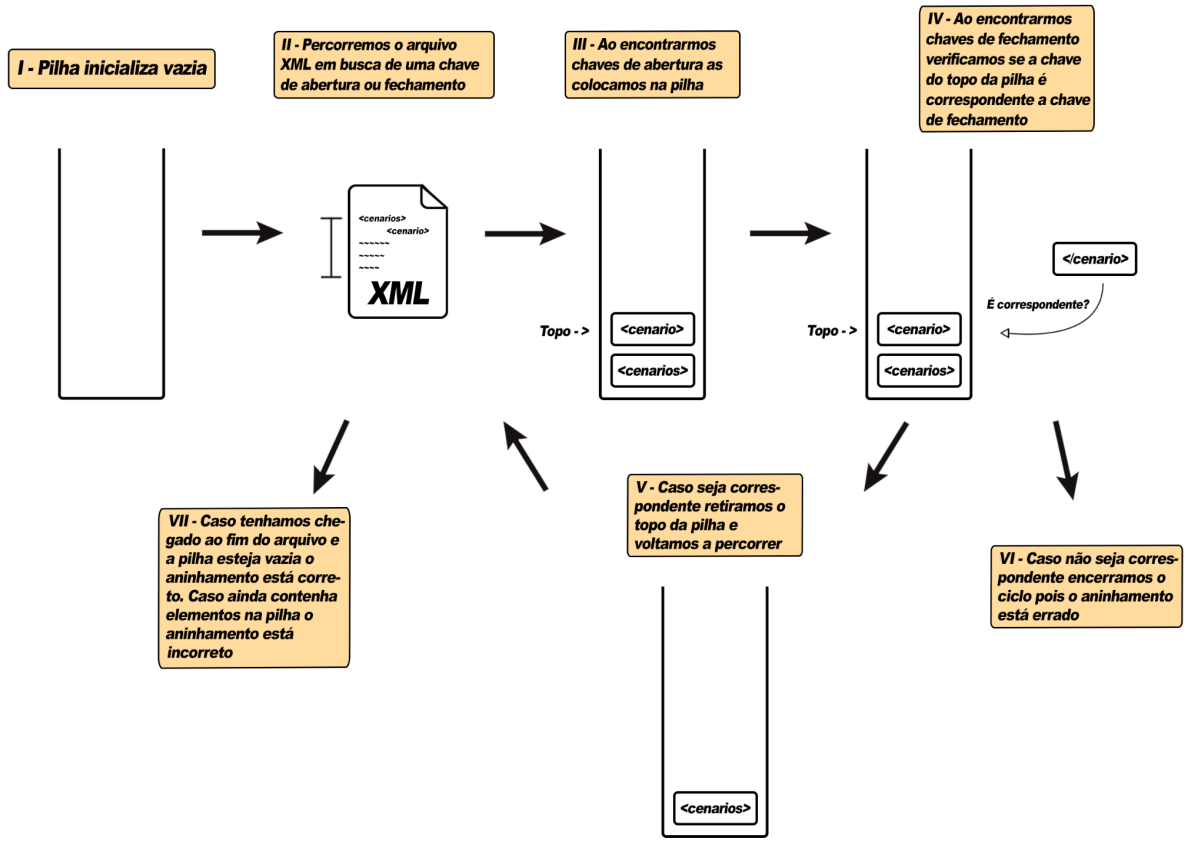
Detalhamento:

1. **Entrada:**
    - A função recebe um argumento, `buffer`, que é um vetor de caracteres contendo o conteúdo do arquivo XML que precisa ser verificado.
2. **Inicialização:**
    - A função inicializa uma pilha (usando a classe `ArrayStack`) para armazenar as tags XML durante o processo de verificação. Também define variáveis para rastrear o estado atual durante a iteração através do `buffer`.

3. **Iteração através do Buffer:**
    - A função itera sobre cada caractere no `buffer`. Quando encontra o caractere '<', assume que uma tag XML está começando e começa a construir a tag completa até encontrar o caractere '>'.
4. **Processamento de Tags:**
    - Se a tag é uma tag de abertura (não começa com '/'), ela é empurrada para a pilha.
    - Se a tag é uma tag de fechamento (começa com '/'), a função remove o '/' e verifica se a tag no topo da pilha corresponde a essa tag de fechamento. Se não corresponder, a função determina que o XML é malformado e define o `resultado` como `false`.
5. **Continuação da Iteração:**
    - A função continua a iterar através do `buffer` até que tenha processado todos os caracteres.
6. **Verificação Final:**
    - Após a iteração, se a pilha não estiver vazia (o que significa que algumas tags de abertura não foram fechadas) ou se não houver nenhuma tag processada, a função considera o XML como mal formado.
7. **Saída:**
    - A função retorna um valor booleano `resultado`, que é `true` se o conteúdo do XML estiver bem formado e `false` se estiver mal formado.

Essas duas primeiras funções do arquivo main.cpp são referentes ao exercício 1, que pedia para verificar o aninhamento de tags em um arquivo xml. A imagem abaixo ilustra como a pilha estava sendo utilizada para a verificação do arquivo xml.

Figura ilustrativa do funcionamento do exercício 1



main.cpp - parte referente ao exercício 2

```
/// @brief Estrutura para armazenar os dados de um cenário
Vitor Calegari, anteontem | 2 authors (You and others)
struct Cenario {
    int** matriz;
    std::pair<int, int> coord_robo, dim_matriz;
    std::string nome;
};
```

Essa estrutura é criada para armazenar dados referentes ao cenário atual do arquivo xml dado como entrada.

main.cpp - parte referente ao exercício 2

```cpp
std::vector<Cenario> vetor_para_matriz(char* buffer) {
    std::vector<Cenario> cenarios;
    char* cenario_ptr = strstr(buffer, "<cenario>"); // Criando ponteiro para a primeira ocorrência de <cenario> do buffer

    // Enquanto houver ocorrências de <cenario> no buffer o laço continua
    while (cenario_ptr) {
        Cenario cenario;

        char* nome_ptr = strstr(cenario_ptr, "<nome>");       // Criando ponteiro para a primeira ocorrência de <nome> do cenário atual
        char* altura_ptr = strstr(cenario_ptr, "<altura>");   // Criando ponteiro para a primeira ocorrência de <altura> do cenário atual
        char* largura_ptr = strstr(cenario_ptr, "<largura>"); // Criando ponteiro para a primeira ocorrência de <largura> do cenário atual
        char* matriz_ptr = strstr(cenario_ptr, "<matriz>");   // Criando ponteiro para a primeira ocorrência de <matriz> do cenário atual
        char* x_ptr = strstr(cenario_ptr, "<x>");             // Criando ponteiro para a primeira ocorrência de <x> do cenário atual
        char* y_ptr = strstr(cenario_ptr, "<y>");             // Criando ponteiro para a primeira ocorrência de <y> do cenário atual

        // Se todos os ponteiros forem válidos, extrair os dados do cenário
        if (nome_ptr && altura_ptr && largura_ptr && matriz_ptr && x_ptr && y_ptr) {
            cenario.nome = std::string(nome_ptr + strlen("<nome>"), strstr(nome_ptr, "</nome>") - (nome_ptr + strlen("<nome>")));  // Extrair
            cenario.dim_matriz.second = atoi(altura_ptr + strlen("<altura>"));     // Extrair a altura do cenário atual
            cenario.dim_matriz.first = atoi(largura_ptr + strlen("<largura>"));  // Extrair a largura do cenário atual
            cenario.coord_robo.first = atoi(x_ptr + strlen("<x>"));  // Extrair a coordenada x do cenário atual
            cenario.coord_robo.second = atoi(y_ptr + strlen("<y>"));  // Extrair a coordenada y do cenário atual

            matriz_ptr += strlen("<matriz>");         // Avançar o ponteiro para o início da matriz
            std::string str_matriz;                   // Criar uma string para armazenar a matriz
```

```cpp
    // Extrair a matriz do cenário atual
    for (int i = 0; i < cenario.dim_matriz.second * cenario.dim_matriz.first; i++) {
        while (*matriz_ptr == '\n' || *matriz_ptr == '\r') {    // Ignorar quebras de linha
                *matriz_ptr++;
        }
        str_matriz.push_back(*matriz_ptr++);                    // Adicionar o caractere atual à string da matriz
    }

    // Alocar memória para a matriz do cenário atual
    cenario.matriz = new int*[cenario.dim_matriz.second];
    for (int i = 0; i < cenario.dim_matriz.second; i++) {
        cenario.matriz[i] = new int[cenario.dim_matriz.first];
    }

    // Preencher a matriz do cenário atual com os dados extraídos
    int k = 0;
    for (int i = 0; i < cenario.dim_matriz.second; i++) {
        for (int j = 0; j < cenario.dim_matriz.first; j++) {
            cenario.matriz[i][j] = str_matriz[k++] - '0';
        }
    }
```

```cpp
            // Adicionar o cenário atual ao vetor de cenários
            cenarios.push_back(cenario);
        }

        // Avançar o ponteiro para a próxima ocorrência de <cenario>
        cenario_ptr = strstr(cenario_ptr + strlen("<cenario>"), "<cenario>");
    }

    return cenarios;
}
```

A função `vetor_para_matriz` é responsável por transformar um vetor de caracteres (buffer) que contém informações sobre vários cenários em um vetor de estruturas `Cenario`. Cada estrutura `Cenario` contém informações sobre um cenário específico, como nome, dimensões da matriz, coordenadas do robô e a matriz em si.

Detalhamento:

1. **Inicialização:**
    - Um vetor vazio `cenarios` é criado para armazenar os cenários extraídos do buffer.
    - Um ponteiro `cenario_ptr` é inicializado para apontar para a primeira ocorrência da tag `<cenario>` no buffer.
2. **Extração dos Cenários:**
    - Um loop `while` é usado para iterar sobre todas as ocorrências da tag `<cenario>` no buffer.
    - Dentro do loop, ponteiros são inicializados para apontar para as tags relevantes dentro do cenário atual, como `<nome>`, `<altura>`, `<largura>`, `<matriz>`, `<x>`, e `<y>`.
3. **Verificação de Ponteiros Válidos:**
    - Se todos os ponteiros forem válidos, o programa prossegue para extrair as informações do cenário atual.
4. **Extração de Informações do Cenário:**
    - O nome, dimensões da matriz, e coordenadas do robô são extraídos usando os ponteiros relevantes e armazenados na estrutura `Cenario`.
    - A matriz é extraída do buffer e armazenada em uma string `str_matriz`.
5. **Conversão da String da Matriz para Matriz 2D:**
    - Memória é alocada para a matriz 2D do cenário atual.
    - A string da matriz é então convertida em uma matriz 2D de inteiros.
6. **Adicionando o Cenário ao Vetor:**
    - O cenário atual é adicionado ao vetor `cenarios`.
7. **Avançando para o Próximo Cenário:**
    - O ponteiro `cenario_ptr` é movido para a próxima ocorrência da tag `<cenario>` no buffer.
8. **Retorno:**
    - Após processar todos os cenários no buffer, o vetor `cenarios` é retornado.

main.cpp - parte referente ao exercício 2

```cpp
void verifica_area(int** matriz, std::pair<int,int> coord_robo,std::pair<int,int> dim_matriz) {
    int x = coord_robo.first;
    int y = coord_robo.second;
    int largura = dim_matriz.first;
    int altura = dim_matriz.second;
    structures::ArrayQueue<std::pair<int, int>> fila; // Criando uma fila de pares de inteiros

    // Alocando memória para a matriz R
    int** R = new int*[altura];
    for (int i = 0; i < altura; i++) {
        R[i] = new int[largura];
    }

    // Inicializando a matriz R com 0s
    for (int linha = 0; linha < altura; linha++) {
        for (int coluna = 0; coluna < largura; coluna++) {
            R[linha][coluna] = 0;
        }
    }

    // Se a posição (x, y) for 0, a área é 0
    if (matriz[x][y] == 0) {
        cout << '0';
        cout << endl;
        return;
    }
```

```cpp
    // Matriz[x][y] = Matriz[i][j]
    fila.enqueue({x, y});
    R[x][y] = 1;

    // Enquanto a fila não estiver vazia
    while (!fila.empty()) {
        auto coord = fila.dequeue();
        x = coord.first;
        y = coord.second;


        // Definindo os vizinhos-4
        std::pair<int, int> vizinhos[4] = {{x, y-1}, {x, y+1}, {x-1, y}, {x+1, y}};

        for (auto& vizinho : vizinhos) {
            int nx = vizinho.first;
            int ny = vizinho.second;


            // Verificar se o vizinho está dentro do domínio da matriz
            if (ny >= 0 && ny < largura && nx >= 0 && nx < altura) {
                // Verificar se o vizinho tem intensidade 1 em E e ainda não foi visitado em R
                if (matriz[nx][ny] == 1 && R[nx][ny] == 0) {
                    fila.enqueue({nx, ny});
                    R[nx][ny] = 1;  // Marcar como visitado
                }
            }
        }
    }
```

```
    // Contar a quantidade de 1s na matriz R para determinar a área do componente conexo
    int area = 0;
    for (int i = 0; i < altura; i++) {
        for (int j = 0; j < largura; j++) {
            if (R[i][j] == 1) {
                area++;
            }
        }
    }

    std::cout << area << std::endl;

    // Liberar memória alocada para R
    for (int i = 0; i < altura; i++) {
        delete[] R[i];
    }
    delete[] R;
}
```

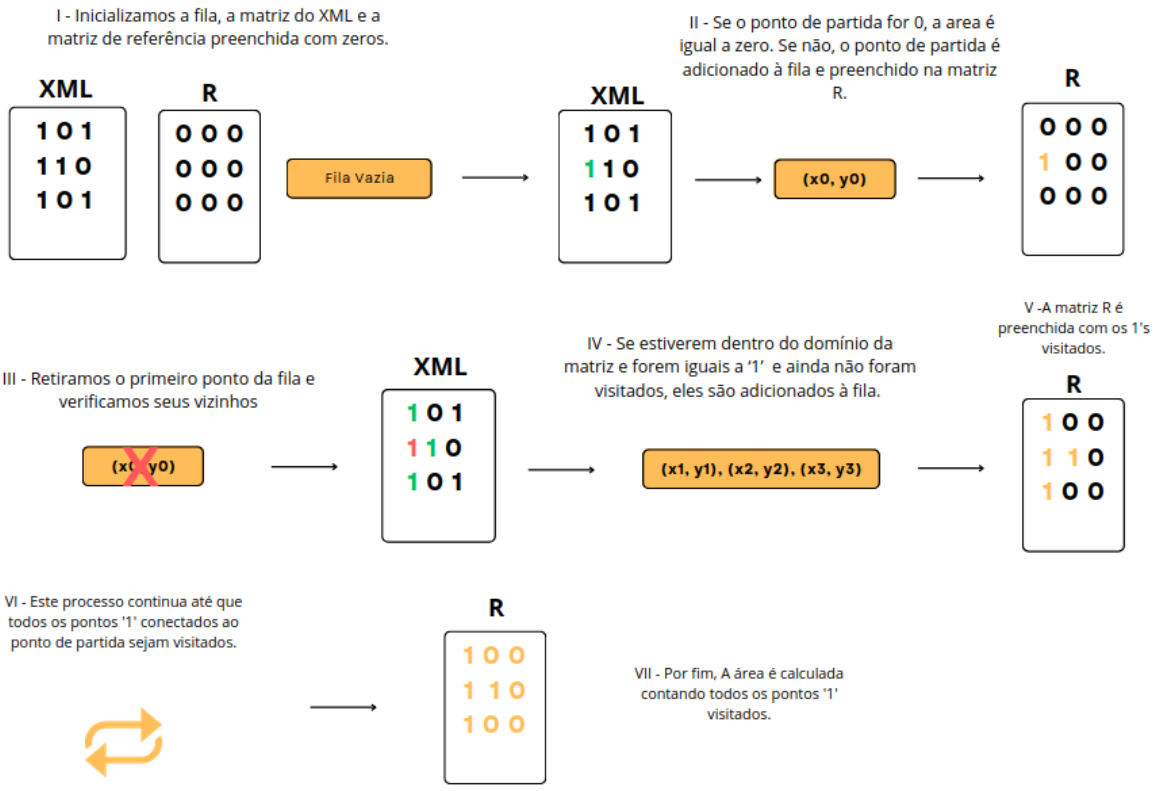
A função `verifica_area` tem como objetivo determinar a área (quantidade de pontos iguais a 1) que está conectada a uma posição inicial dada na matriz, usando o conceito de vizinhança-4.

Detalhamento:

1. **Inicialização de Variáveis:**
   - As coordenadas do robô e as dimensões da matriz são extraídas dos pares de inteiros passados como argumentos.
   - Uma fila de pares de inteiros é criada para armazenar as coordenadas que precisam ser verificadas.
   - Uma matriz `R` é alocada e inicializada com zeros. Esta matriz servirá para marcar quais posições já foram visitadas.
2. **Verificação de Posição Inicial:**
   - Se a posição inicial (coordenadas do robô) na matriz for 0, a função retorna imediatamente com uma área de 0.
3. **Início da Verificação:**
   - A posição inicial é adicionada à fila e marcada como visitada na matriz `R`.
4. **Loop Principal:**
   - Enquanto a fila não estiver vazia, a função continua verificando as posições.
   - A posição atual é desenfileirada e seus vizinhos-4 são determinados.
   - Para cada vizinho, a função verifica se ele está dentro dos limites da matriz, se tem valor 1 na matriz original e se ainda não foi visitado na matriz `R`.
   - Se todas essas condições forem verdadeiras, o vizinho é adicionado à fila e marcado como visitado na matriz `R`.

5. **Cálculo da Área:**
    - Após o loop principal, a função conta quantos 1s existem na matriz R. Esse número representa a área do componente conexo.
6. **Liberação de Memória:**
    - A memória alocada para a matriz R é liberada.

Figura ilustrativa do funcionamento do exercício 2



main.cpp - função main

```cpp
int main() {
    char xmlfilename[100];
    char * buffer;

    // Recebe o nome do arquivo a ser lido
    cin >> xmlfilename;

    // Lê o arquivo e guarda os dados em um buffer
    buffer = arquivo_para_vetor(xmlfilename);

    // Verifica se o buffer segue a regra de aninhamento de arquivos .xml
    if (!verifica_vetor(buffer)) {
        cout << "erro";
        delete[] buffer;
        return 0;
    }

    // Extrai os dados dos cenários e armazena em um vetor de estruturas Cenario
    std::vector<Cenario> cenarios = vetor_para_matriz(buffer);

    // Para cada cenário, verificar a área do componente conexo que contém a posição (x, y) na matriz
    for (const Cenario& cenario : cenarios) {
        cout << cenario.nome << " ";
        verifica_area(cenario.matriz, cenario.coord_robo, cenario.dim_matriz);

        // Libera a memória alocada pela matriz do cenário atual
        for (int i = 0; i < cenario.dim_matriz.second; i++) {
            delete[] cenario.matriz[i];
        }
        delete[] cenario.matriz;
    }

    delete[] buffer;
    cout << endl;

    return 0;
}
```

A função `main` serve como ponto de entrada para o programa, coordenando as operações de leitura, processamento e saída dos dados.

1. **Inicialização de Variáveis:**
    - `xmlfilename` é uma string que armazenará o nome do arquivo XML a ser lido.
    - `buffer` é um ponteiro que armazenará os dados lidos do arquivo.

2. **Entrada de Dados:**
    - O nome do arquivo XML é lido do `cin` e será armazenado em `xmlfilename`.
3. **Leitura do Arquivo:**

    - A função `arquivo_para_vetor` é chamada para ler o arquivo e armazenar seu conteúdo no `buffer`.
4. **Verificação de Aninhamento:**
    - A função `verifica_vetor` é chamada para verificar se o conteúdo do `buffer` segue as regras de aninhamento de arquivos XML.
    - Se não seguir, o programa imprime "erro", libera a memória alocada para o `buffer` e termina.
5. **Extração dos Dados dos Cenários:**
    - A função `vetor_para_matriz` é chamada para extrair os dados dos cenários do `buffer` e armazená-los em um vetor de estruturas `Cenario`.
6. **Processamento de Cada Cenário:**
    - Para cada cenário no vetor de cenários:
        - O nome do cenário é impresso.
        - A função `verifica_area` é chamada para determinar a área do componente conexo que contém a posição inicial do robô na matriz do cenário.
        - A memória alocada para a matriz do cenário é liberada.
7. **Finalização:**
    - A memória alocada para o `buffer` é liberada.
    - Uma quebra de linha é impressa.
    - O programa termina com sucesso.

array_queue.h

```
//! classe ArrayQueue
class ArrayQueue {
 public:
    //! construtor padrao
    ArrayQueue();
    //! construtor com parametro
    explicit ArrayQueue(std::size_t max);
    //! destrutor padrao
    ~ArrayQueue();
    //! metodo enfileirar
    void enqueue(const T& data);
    //! metodo desenfileirar
    T dequeue();
    //! metodo retorna o ultimo
    T& back();
    //! metodo limpa a fila
    void clear();
    //! metodo retorna tamanho atual
    std::size_t size();
    //! metodo retorna tamanho maximo
    std::size_t max_size();
    //! metodo verifica se vazio
    bool empty();
    //! metodo verifica se esta cheio
    bool full();

 private:
    T* contents;
    std::size_t size_;
    std::size_t max_size_;
    int begin_;  // indice do inicio (para fila circular)
    int end_;  // indice do fim (para fila circular)
    static const auto DEFAULT_SIZE = 1000u;
};

}  // namespace structures


template<typename T>
structures::ArrayQueue<T>::ArrayQueue() {
    max_size_ = DEFAULT_SIZE;
    contents = new T[max_size_];
    size_ = 0;
    begin_ = 0;
    end_ = -1;
}
```

```cpp
template<typename T>
structures::ArrayQueue<T>::ArrayQueue(std::size_t max) {
    max_size_ = max;
    contents = new T[max];
    size_ = 0;
    begin_ = 0;
    end_ = -1;
}

template<typename T>
structures::ArrayQueue<T>::~ArrayQueue() {
    delete [] contents;
}

template<typename T>
void structures::ArrayQueue<T>::enqueue(const T& data) {
    if (full()) {
        throw std::out_of_range("fila cheia");
    } else {
        end_ = (end_+1)%max_size_;
        contents[end_] = data;
        size_++;
    }
}

template<typename T>
T structures::ArrayQueue<T>::dequeue() {
    if (empty()) {
       throw std :: out_of_range("fila vazia");
    } else {
       T aux;
       aux = contents[begin_];
       size_--;
       begin_ = (begin_+1)%max_size_;
       return aux;
    }
}

template<typename T>
T& structures::ArrayQueue<T>::back() {
    if (empty()) {
       throw std :: out_of_range("fila vazia");
    } else {
       return contents[end_];
    }
}
```

```cpp
template<typename T>
void structures::ArrayQueue<T>::clear() {
     end_ = -1;
     size_ = 0;
     begin_ = 0;
}

template<typename T>
std::size_t structures::ArrayQueue<T>::size() {
     return size_;
}

template<typename T>
std::size_t structures::ArrayQueue<T>::max_size() {
    return max_size_;
}

template<typename T>
bool structures::ArrayQueue<T>::empty() {
    if (size_ == 0) {
      return true;
    } else {
      return false;
    }
}

template<typename T>
bool structures::ArrayQueue<T>::full() {
    if (size_ == max_size_) {
      return true;
    } else {
      return false;
    }
}

#endif
```

Esse arquivo contém a classe fila que foi usada para implementar a lógica do exercício 2.

array_stack.h

```
//! CLASSE PILHA
class ArrayStack {
 public:
    //! construtor simples
    ArrayStack();
    //! construtor com parametro tamanho
    explicit ArrayStack(std::size_t max);
    //! destrutor
    ~ArrayStack();
    //! metodo empilha
    void push(const T& data);
    //! metodo desempilha
    T pop();
    //! metodo retorna o topo
    T& top();
    //! metodo limpa pilha
    void clear();
    //! metodo retorna tamanho
    std::size_t size();
    //! metodo retorna capacidade maxima
    std::size_t max_size();
    //! verifica se esta vazia
    bool empty();
    //! verifica se esta cheia
    bool full();

 private:
    T* contents;
    int top_;
    std::size_t max_size_;

    static const auto DEFAULT_SIZE = 10u;
};

}  // namespace structures

#endif


template<typename T>
structures::ArrayStack<T>::ArrayStack() {
    max_size_ = DEFAULT_SIZE;
    contents = new T[max_size_];
    top_ = -1;
}
```

```cpp
template<typename T>
structures::ArrayStack<T>::ArrayStack(std::size_t max) {
    max_size_ = max;
    contents = new T[max_size_];
    top_ = -1;
}

template<typename T>
structures::ArrayStack<T>::~ArrayStack() {
    delete [] contents;
}

template<typename T>
void structures::ArrayStack<T>::push(const T& data) {
    if (full()) {
        throw std::out_of_range("pilha cheia");
    } else {
        top_++;
        contents[top_] = data;
    }
}

template<typename T>
T structures::ArrayStack<T>::pop() {
    if (empty()) {
        throw std::out_of_range("pilha vazia");
    } else {
        int index_retorno = top_;
        top_--;
        return contents[index_retorno];
    }
}

template<typename T>
T& structures::ArrayStack<T>::top() {
    return contents[top_];
}

template<typename T>
void structures::ArrayStack<T>::clear() {
    top_ = -1;
}

template<typename T>
std::size_t structures::ArrayStack<T>::size() {
    int index_retorno = top_ + 1;
    return index_retorno;
}
```

```cpp
template<typename T>
std::size_t structures::ArrayStack<T>::max_size() {
    return max_size_;
}

template<typename T>
bool structures::ArrayStack<T>::empty() {
    if (top_ == -1) {
        return true;
    } else {
        return false;
    }
}

template<typename T>
bool structures::ArrayStack<T>::full() {
    if ((top_ + 1) == static_cast<int>(max_size_)) {
        return true;
    } else {
        return false;
    }
}
```

Esse arquivo contém a classe pilha que foi usada para implementar a lógica do exercício 1.

3 - Conclusão

Ao longo do desenvolvimento deste projeto, enfrentamos uma série de desafios que nos proporcionaram um aprendizado profundo e uma compreensão mais aprimorada sobre a importância da precisão na programação. A sintaxe correta, por exemplo, mostrou-se crucial. Pequenos deslizes ou inversões, como a ordem de coordenadas (x, y) em comparação a (y, x), podem levar a resultados completamente diferentes e, muitas vezes, inesperados.

Entender os algoritmos foi outra etapa que exigiu dedicação. A lógica por trás de cada função e a maneira como os dados são processados e armazenados são fundamentais para garantir a eficiência e a precisão do programa. Além disso, a implementação de estruturas de dados, como pilhas e filas, reforçou a importância de compreender plenamente as operações e os conceitos associados a essas estruturas.

Achar erros no código, ou "debugging", foi talvez uma das tarefas mais desafiadoras. Erros de segmentação, em particular, podem ser difíceis de rastrear, pois muitas vezes a causa raiz não está imediatamente aparente. Isso nos ensinou a importância de uma abordagem sistemática para a depuração e a necessidade de testar o código regularmente, em diferentes etapas do desenvolvimento.

Em resumo, este projeto não foi apenas uma oportunidade para aplicar conceitos teóricos, mas também uma experiência valiosa em resolução de

problemas e atenção aos detalhes. Estas são habilidades essenciais para qualquer programador e serão inestimáveis em futuros projetos e desafios.

4 - Referências

https://en.cppreference.com/w/
https://stackoverflow.com/