# Relatório do Projeto
# Sistema Hospitalar em C++

**Disciplina:** Estrutura de Dados Orientados a Objetos

**Curso/Período:** Sistemas de Informação/2025.2

**Professor:** Francisco Paulo Magalhães Simões

**Grupo:** Ana Laura Barboza Oliveira dos Santos

Caio Cesar Nascimento Vilas Boas

Eduardo Alves Pinto Vilar de Oliveira

Letícia Staudinger Ribeiro

Safira Moraes Gomes

## 1. Introdução

Este projeto é um sistema hospitalar em C++ voltado para o gerenciamento de pacientes, médicos e consultas. O objetivo é melhorar a organização, automação de processos e controle de dados desse tipo de ambiente, tornando o controle de fila de atendimento e o fluxo de informações efetivo. Além disso, o sistema oferece operações completas de CRUD (criação, leitura, atualização e remoção) para pacientes, médicos e consultas, permitindo controle total sobre os registros e garantindo que as informações se mantenham consistentes durante toda a execução.

O desenvolvimento foi feito seguindo os princípios da Programação Orientada a Objetos (POO), permitindo uma estrutura modular, organizada e de fácil manutenção Também utiliza arquivos JSON para armazenamento das informações.

## 2. Fundamentação teórica e implementação

A Programação Orientada a Objetos (POO) foi utilizada como base para o desenvolvimento do projeto. Foram aplicados conceitos como encapsulamento, garantindo que os atributos das classes fossem acessados apenas através de métodos específicos; herança, utilizada para criar classes especializadas a partir de uma classe base, como *Paciente* e *Medico* herdando de *Pessoa*; polimorfismo, permitindo que métodos com o mesmo nome se comportassem de formas diferentes conforme a classe do objeto; e modularização, separando responsabilidades entre diferentes classes (Paciente, Medico, Consulta, Hospital etc.). O projeto também conta com construtores e destrutores para inicialização e liberação de recursos, garantindo a integridade da memória, especialmente por lidar com ponteiros.

Estruturas da Standard Template Library (STL), como vetores, listas, fila e strings, foram utilizadas para armazenamento e manipulação de dados. Listas foram usadas para manter registros de pacientes, médicos e consultas, enquanto a fila de prioridade (priority_queue) implementou a lógica de atendimento segundo prioridades médicas. Vetores e strings facilitaram o manuseio de dados temporários e de interface com o usuário. Ademais, o projeto faz uso de um arquivo JSON para simular um banco de dados, armazenando informações persistentes como pacientes, médicos e consultas entre diferentes execuções do sistema.

O fluxo de execução usa múltiplas filas de prioridade (<*code*>*priority_queue*</*code*>), onde cada médico possui sua própria instância de *FilaAtendimento*. A fila não armazena pacientes, e sim ponteiros para *Consulta*. Isso permite um sistema de prioridade de três níveis:

1. **Prioridade de Triagem** (da consulta, ex: Emergência);
2. **Prioridade de Vulnerabilidade** (do paciente, ex: Alto Risco);
3. **Ordem de Chegada** (para desempate).

Ao atender, o usuário **seleciona qual médico** irá chamar o próximo paciente, e o sistema busca a consulta mais prioritária da fila *daquele* médico específico.

## 3. Análise e planejamento do sistema

O sistema foi planejado para oferecer funcionalidades essenciais de um ambiente hospitalar básico, como cadastro de pacientes e médicos, agendamento de consultas, relatórios e gerenciamento de uma fila de atendimento.

Os requisitos funcionais foram:

- Agendar consultas (e enfileirar automaticamente na fila do médico);
- Atender o próximo paciente (selecionando um **médico específico** para o atendimento);
- Cancelar uma consulta agendada;
- Consultar relatórios de consultas feitas por um médico, consultas de um paciente e tempo médio de espera na fila.

CRUD (Create, Read, Update, Delete):

- Cadastrar pacientes e médicos;
- Leitura de pacientes, médicos e consultas armazenadas no sistema;
- Edição de pacientes, médicos e consultas com atualização de dados já existentes;
- Remoção de pacientes, médicos e consultas, respeitando dependências (não é permitido excluir pacientes ou médicos com consultas ativas);
- Persistência automática das alterações no arquivo JSON;
- Recarregamento automático dos dados ao iniciar o programa.

E os requisitos não funcionais, por sua vez, foram:

- Modularização do código;
- Simplicidade de uso;
- Persistência de dados via JSON.

## 4. Estrutura do código

O código foi organizado em múltiplos arquivos, seguindo o padrão de separação entre declaração (arquivos .h) e implementação (arquivos .cpp). Abaixo, segue a estruturação de classes, assim como seus atributos e métodos.

**Classe Pessoa:** definição de atributos e métodos básicos de qualquer pessoa.
Atributos: nome (string), idade (int)
Métodos:

- Construtor Pessoa(nome, idade);
- Destrutor virtual ~Pessoa();
- Setters: setNome(nome), setIdade(idade);
- Getters: getNome(), getIdade();
- toJSONString(): serializa objeto para JSON.

**Classe Paciente (herda Pessoa):** representa pacientes do hospital.
Atributos: prioridadeVulnerabilidade (int), historicoMedico (string)
Métodos:

- Construtor Paciente(nome, idade, **prioridadeVulnerabilidade**, historico);
- Setters: **setPrioridadeVulnerabilidade(p)** , setHistorico(historico);
- Getters: **getPrioridadeVulnerabilidade()** , getHistorico().

**Classe Medico (herda Pessoa):** representa médicos do hospital.
Atributos: crm (string), especialidade (string), FilaAtendimento filaDeAtendimento
Métodos:

- Construtor Medico(nome, idade, crm, especialidade);
- Setters: setCRM(crm), setEspecialidade (especialidade);
- Getters: getCRM(), getEspecialidade(), getFila();
- toJSONString() e fromJSONString(jsonStr): serialização JSON.

**Classe Consulta:** representa consultas agendadas entre pacientes e médicos.
Atributos: id (int), paciente (Paciente*), medico (Medico*), data (string), status (string), prioridadeTriagem (int) , ordemChegada (long long)
Métodos:

- Construtor Consulta(id, paciente, medico, data, int prioridadeTriagem);
- concluirConsulta(), cancelarConsulta();
- Getters: getId(), getPaciente(), getMedico(), getData(), getStatus();
- toJSONString() e fromJSONString(jsonStr, hospital).

**Struct ComparadorPaciente:** Struct ComparadorPaciente: define a comparação de **Consultas** para a fila de prioridade.
Método:

- *operador()(const Consulta* a, const Consulta* b)* que implementa a lógica de **três níveis** (Triagem, Vulnerabilidade e Ordem de Chegada).

**Classe FilaAtendimento:** gerencia a fila de consultas com prioridade
Atributos: fila (std::priority_queue<Consulta*, vector<Consulta*>, ComparadorPaciente>) • proximaOrdem (long long)
Métodos:

- adicionarConsulta(Consulta*);
- chamarProximo();
- estaVazia();
- tamanho();
- visualizarFila().

**Classe Hospital:** centraliza cadastros, consultas e o gerenciamento completo das entidades (pacientes, médicos e consultas).
Atributos: pacientes (list<Paciente*>), medicos (list<Medico*>), consultas (list<Consulta*>), proximoIdConsulta (int)
Métodos:

- Construtor Hospital(), Destrutor ~Hospital();
- Cadastros: cadastrarPaciente(...), cadastrarMedico(...);
- Consultas/Fila: agendarEEnfileirar(...) , atenderProximo(std::string nomeMedico), cancelarConsulta(int consultaId);
- Buscas: buscarPacientePorNome(nome), buscarMedicoPorNome(nome), buscarConsultaPorId(int id);
- Edição (CRUD - Update): editarPaciente(nomeAtual), editarMedico(nomeAtual), editarConsulta(consultaId) - Permitem atualizar dados das entidades com validação dos campos e preservação de informações sensíveis (como histórico ou vínculos de consulta);
- Remoção (CRUD - Delete): removerPaciente(nome), removerMedico(nome), removerConsulta(consultaId) – As remoções possuem verificação de dependências: um paciente ou médico só pode ser excluído se não possuir consultas ativas (Agendadas ou Concluídas);
- Validação: pacientePossuiConsultas(nomePaciente), medicoPossuiConsultas(nomeMedico);
- Listagens: listarPacientes(), listarMedicos(), listarConsultas(), listarFilaAtendimento();
- Persistência JSON: salvarDados(arquivo), carregarDados(arquivo);
- Auxiliares: getTotalPacientes(), getTotalMedicos(), getTotalConsultas(), getConsultas().

**Namespace Relatorios:** funções para gerar relatórios sobre consultas, pacientes e médicos.
Funções:

- gerarRelatorioMedicos(consultas);
- gerarRelatorioTempoMedio(consultas);
- gerarHistoricoPorPaciente(consultas, nome);
- gerarHistoricoPorMedico(consultas, nome);
- testarRelatorios(consultas).

**main.cpp:** interface de console que interage com a classe Hospital. O arquivo principal do programa implementa menus interativos e funções auxiliares que permitem executar todas as operações CRUD. Ele possui submenus específicos para Pacientes, Médicos, Consultas, Relatórios e Persistência, organizando as funcionalidades de forma clara.
Funções auxiliares:

- limparBuffer() - esvazia o buffer de entrada para evitar leituras incorretas;

- lerInteiro(mensagem) - lê números inteiros com validação;
- lerString(mensagem) - lê strings completas;
- configurarUTF8() - garante compatibilidade de acentuação em Windows e Linux.

Submenus:

- Pacientes - cadastrar, listar, editar e remover;
- Médicos - cadastrar, listar, editar e remover;
- Consultas - agendar, editar, cancelar e remover;
- Relatórios - gerar históricos e estatísticas de atendimento;
- Persistência - salvar e carregar dados manualmente, além de salvar automaticamente ao encerrar o sistema.

**Fluxo principal:** controla as chamadas aos métodos do Hospital e mantém o programa ativo até a escolha da opção de saída. O main.cpp também integra a persistência de dados, carregando automaticamente as informações ao iniciar e salvando-as antes de encerrar, garantindo consistência entre execuções.

### 4.1. Tabela da Estruturação do Código

| **Classe / Estrutura** | **Tipo / Papel** | **Herança** | **Depende de** | **Usado por** | **Relacionamento Principal** |
|---|---|---|---|---|---|
| Pessoa | Classe base | – | – | Paciente, Medico | Classe-mãe |
| Paciente | Classe derivada | Pessoa | – | Consulta, FilaAtendimento, Hospital | Herança e composição |
| Medico | Classe derivada | Pessoa | – | Consulta, Hospital | Herança e composição |
| Consulta | Classe de domínio | – | Paciente, Medico, Hospital | Hospital, Relatorios | Associação (ponteiros) |
| ComparadorPaciente | Struct auxiliar | – | Paciente | FilaAtendimento | Functor de comparação |
| FilaAtendimento | Classe de controle | – | Paciente, ComparadorPaciente | Médico | Composição |
| Hospital | Classe gerenciadora | – | Paciente, Medico, Consulta, Relatorios | – | Composição total |
| Relatorios | Namespace | – | Consulta | Hospital | Associação funcional |
| nlohmann::json | Biblioteca externa | – | Todas (persistência) | – | Integração externa |

## 5. Banco de dados (JSON)

Para simplificar a persistência de dados, foi utilizado um arquivo JSON. Essa abordagem permite armazenar informações estruturadas sem a necessidade de um servidor de banco de dados. As operações de leitura e escrita são realizadas por meio da biblioteca nlohmann/json, que facilita a serialização e desserialização de objetos C++ para o formato JSON. Cada classe que armazena dados relevantes implementa métodos de serialização e desserialização:

- **Pessoa::toJSONString():** Converte os atributos básicos de uma pessoa (nome e idade) para uma string JSON, servindo como base para heranças (como Paciente e Médico), que acrescentam seus próprios campos;
- **Paciente::toJSONString():** Estende a serialização da Pessoa, incluindo prioridadeVulnerabilidade e historicoMedico.
- **Paciente::fromJSONString():** Recebe uma string JSON e recria um objeto Paciente com os dados correspondentes. Permite carregar pacientes do arquivo JSON de forma dinâmica;
- **Medico::toJSONString():** Serializa os dados do médico, incluindo crm e especialidade, além dos atributos herdados de Pessoa;
- **Medico::fromJSONString():** Constrói um objeto Médico a partir de uma string JSON, garantindo que todos os campos sejam corretamente restaurados;
- **Consulta::toJSONString():** Serializa as consultas, armazenando id, paciente, médico, data, status e prioridadeTriagem. Para paciente e médico, apenas os nomes são salvos, já que os objetos completos estão gerenciados pelo hospital;
- **Consulta::fromJSONString():** Reconstrói uma consulta a partir do JSON. Para isso, utiliza o Hospital como referência para buscar os objetos Paciente* e Medico* correspondentes aos nomes armazenados. Isso garante consistência ao carregar os dados.

No nível do Hospital, a persistência é centralizada em dois métodos principais:

**a) Hospital::salvarDados(const std::string& arquivo):**

- Cria um objeto JSON contendo três arrays principais: pacientes, médicos e consultas;
- Para cada lista, itera sobre os ponteiros e utiliza o método toJSONString() da respectiva classe para gerar a representação JSON;
- Também salva o valor de proximoIdConsulta para manter a contagem correta ao reiniciar o sistema;
- Escreve o JSON em um arquivo (hospital_data.json por padrão) com identação de 4 espaços para legibilidade;
- É chamado tanto manualmente pelo usuário (opção de menu “Salvar Dados”) quanto automaticamente ao sair do programa.

**b) Hospital::carregarDados(const std::string& arquivo):**

- Abre o arquivo JSON, parseia seu conteúdo e limpa as listas internas do hospital (liberando memória de objetos antigos);

- Itera sobre cada array (pacientes, médicos, consultas) e usa os métodos fromJSONString() para recriar os objetos correspondentes;
- Para consultas, garante que os ponteiros para pacientes e médicos sejam recuperados corretamente. **Além disso, o método re-enfileira automaticamente as consultas com status 'Agendada' na fila do médico correspondente** .
- Atualiza proximoIdConsulta para que novas consultas recebam IDs sequenciais corretos;
- É chamado automaticamente na inicialização do programa para restaurar o estado anterior.

O **main.cpp,** por sua vez, integra a persistência ao fluxo do sistema:

- Ao iniciar, chama hospital.carregarDados() para restaurar pacientes, médicos e consultas previamente salvos;
- Antes de encerrar (opção 0 do menu), chama hospital.salvarDados() para garantir que todas as alterações sejam mantidas;
- Há também uma opção de menu (5) para salvar dados manualmente a qualquer momento.

## 6. Testes e validação

Foram realizados testes unitários e de integração, verificando o funcionamento dos métodos de cadastro, agendamento, atendimento e relatórios. Durante os testes, observou-se o correto funcionamento do ciclo de vida das consultas e o registro adequado das informações. Também foram testadas as novas rotinas de edição e remoção, verificando se as alterações realizadas nos registros de pacientes, médicos e consultas eram refletidas corretamente no arquivo JSON e nas listagens subsequentes.

## 7. Resultados e conclusões

O sistema hospitalar apresenta um fluxo completo desde o cadastro até o atendimento dos pacientes, com uso de POO em C++ e organização clara das entidades principais, além de um conjunto de funcionalidades de CRUD para todas as entidades, possibilitando criação, leitura, atualização e remoção de registros de forma controlada. Dentre elas, funcionalidades de cadastro de pacientes e médicos, agendamento de consultas, gerenciamento de filas de atendimento e registro do histórico médico, garantindo um fluxo completo de interação dentro do hospital.

Para persistência de dados, foi utilizado um arquivo JSON, permitindo salvar e carregar informações de pacientes, médicos e consultas de forma estruturada. O sistema também inclui métodos de serialização e desserialização de objetos, garantindo que o estado do hospital seja mantido entre sessões. Foram realizados testes de unidade e integração, confirmando que os métodos funcionam corretamente e que a lógica de atendimento segue a sequência adequada, do cadastro ao atendimento e registro no histórico. No geral, o sistema demonstrou funcionamento consistente, integrando cadastro, agendamento, fila de atendimento e registro de consultas de maneira eficiente.

## 8. Contribuições dos integrantes

Ana Laura Barboza Oliveira dos Santos: classe Relatorio; testes; Read Me do GitHub.
Caio Cesar Nascimento Vilas Boas: main; persistência de dados JSON.
Eduardo Alves Pinto Vilar de Oliveira: classes Pessoa, Paciente e Medico; Makefile.
Letícia Staudinger Ribeiro: classe Hospital; relatório (entrega); testes e correções; GitHub Page.
Safira Moraes Gomes: classes Consulta e FilaAtendimento; vídeo para o Youtube.