

oi deepseek, recentemente eu estava trabalhando em uma classe java pra criar queryparameters a partir de uma classe usando reflection. Aí me veio uma ideia muito louca. E se eu criasse um executável, um helper que lerá código fonte c++ e cria um header file ou inclua no cpp file código para serializar e deserializar aquele objeto. Esse executável iria percorrer o projeto lendo o arquivo do cmake e aí encontraria arquivos com um preprocessador igual tem o qt com QOBJECT acho. e ai geraria no arquivo isso.

Sua ideia é excelente e tem muita similaridade com sistemas de reflexão/metaprogramação já existentes no ecossistema C++! Vou te dar uma visão detalhada de como implementar essa ferramenta:

Arquitetura da Ferramenta

1. Estrutura Geral:

```
bash

cpp-serializer-gen/
└── parser/          # Análise de código C++
└── generator/       # Geração de código
└── cmake_parser/    # Leitura de CMakeLists.txt
└── main.cpp
```

2. Marcador Customizado (como Q_OBJECT):

```
cpp

// Exemplo de uso no código do usuário
class MinhaClasse {
    SERIALIZABLE(MinhaClasse) // Seu marcador customizado

public:
    int id;
    std::string nome;
    std::vector<double> valores;

private:
    // Campos privados também podem ser serializados
```

```
    std::string segredo;  
};
```

3. Implementação do Parser:

cpp

```
class CppParser {  
public:  
    struct ClassInfo {  
        std::string className;  
        std::vector<FieldInfo> fields;  
        std::string sourceFile;  
        bool hasSerializableMacro;  
    };  
  
    std::vector<ClassInfo> parseFile(const std::string& filename) {  
        // Usar Clang AST ou parser customizado  
        // Buscar por SERIALIZABLE macro  
    }  
};
```

4. Geração de Código:

cpp

```
class CodeGenerator {  
public:  
    std::string generateSerializer(const ClassInfo& cls) {  
        return R"(  
template<typename Archive>  
void serialize(Archive& ar, )" + cls.className + R"(& obj) {  
    )" + generateFieldSerialization(cls.fields) + R"(  
}  
    );  
}  
  
    std::string generateDeserializer(const ClassInfo& cls) {  
        // Código similar para deserialização  
    }  
};
```

5. Integração com CMake

• Integração com CMake.

cpp

```
class CMakeParser {
public:
    std::vector<std::string> findSourceFiles(
        const std::string& cmakeFile) {
        // Parse CMakeLists.txt para encontrar:
        // - add_executable()
        // - add_library()
        // - target_sources()
        // E extrair lista de arquivos .cpp/.h
    }
};
```

Formas de Integração:

Opção A: Pré-processamento (Recomendado)

cmake

```
# No CMakeLists.txt do usuário
find_program(SERIALIZER_GEN cpp-serializer-gen)
add_custom_command(
    OUTPUT ${GENERATED_FILES}
    COMMAND ${SERIALIZER_GEN}
    ARGS --input ${SOURCE_FILES}
        --output-dir ${CMAKE_CURRENT_BINARY_DIR}/generated
    DEPENDS ${SOURCE_FILES}
)
```

Opção B: Injeção Direta nos Arquivos

bash

```
# Modifica os arquivos .cpp/.h originais
cpp-serializer-gen --inject --project-dir .
```

Opção C: Arquivo de Configuração

yaml

```
# .serializer-config.yaml
targets:
- name: MyLibrary
  sources:
    - src/**/*.cpp
    - include/**/*.h
  output: generated/
  formats: [json, binary, xml]

serialization:
  include_private: true
  custom_macros: [SERIALIZABLE, PACKED_OBJECT]
```

Exemplo de Código Gerado:

cpp

```
// Arquivo gerado: MinhaClasse_serialization.h
#pragma once
#include "MinhaClasse.h"
#include <nlohmann/json.hpp>

namespace serialization {
    template<>
    struct Serializer<MinhaClasse> {
        static nlohmann::json to_json(const MinhaClasse& obj) {
            return {
                {"id", obj.id},
                {"nome", obj.nome},
                {"valores", obj.valores},
                {"segredo", obj.getSegredo()} // Usa getter se disponível
            };
        }

        static MinhaClasse from_json(const nlohmann::json& j) {
            MinhaClasse obj;
            obj.id = j["id"];
            obj.nome = j["nome"];
            obj.valores = j["valores"].get<std::vector<double>>();
            // setters privados se necessário
            return obj;
        }
    };
}
```

Tecnologias Recomendadas:

1. **Clang LibTooling** - Para parsing robusto de C++
2. **LLVM** - Para manipulação de AST
3. **nlohmann/json** - Como biblioteca de serialização base
4. **Boost Spirit** - Alternativa para parsing customizado
5. **YAML-CPP** - Para arquivos de configuração

Desafios e Considerações:

1. **Dependências Circulares** - Precisa lidar com forward declarations
2. **Templates** - Serialização de classes template é complexa
3. **Herança** - Precisa suportar hierarquia de classes
4. **Performance** - O parsing de AST pode ser lento para projetos grandes
5. **Incremental** - Regerar apenas o que mudou

Alternativas Existentes:

- **Google Protobuf** - Já faz algo similar com .proto files
- **Boost Serialization** - Mas requer código manual
- **RTTR (Run Time Type Reflection)** - Reflexão em tempo de execução
- **Meta** - Biblioteca de reflexão estática

Sua ideia é viável e bastante útil! A parte mais complexa será o parser de C++ robusto. Começar com um subset da linguagem (sem templates, herança múltipla) e expandir gradualmente é uma boa estratégia.

Quer que eu detalhe alguma parte específica da implementação?

me fale das alternativas existentes e exemplifique com código

Alternativas Existentes para Serialização/Reflexão em C++

Vou detalhar as principais alternativas com exemplos práticos:

1. Google Protocol Buffers (mais próximo da sua ideia)

Arquivo .proto (DSL):

```
protobuf

syntax = "proto3";

message Pessoa {
    int32 id = 1;
    string nome = 2;
    string email = 3;
    repeated string telefones = 4;

    enum Tipo {
        PESSOA_FISICA = 0;
        PESSOA_JURIDICA = 1;
    }
    Tipo tipo = 5;
}

message Endereco {
    string rua = 1;
    string cidade = 2;
    string cep = 3;
}
```

Geração do código (similar ao que você quer fazer):

```
bash

# Gera os headers/source files
protoc --cpp_out=. pessoa.proto
```

Código gerado (pessoa.pb.h):

```
cpp

// Código C++ gerado automaticamente
class Pessoa : public ::google::protobuf::Message {
public:
```

```
// Getters e setters automáticos
int32_t id() const;
void set_id(int32_t value);

const std::string& nome() const;
void set_nome(const std::string& value);

// Serialização/Deserialização
bool SerializeToString(std::string* output) const;
bool ParseFromString(const std::string& data);

// JSON
google::protobuf::util::MessageToJsonString(...);
};
```

Uso:

cpp

```
#include "pessoa.pb.h"

Pessoa pessoa;
pessoa.set_id(123);
pessoa.set_nome("João Silva");

// Serializa para string
std::string serializado;
pessoa.SerializeToString(&serializado);

// Deserializa
Pessoa pessoa2;
pessoa2.ParseFromString(serializado);

// Para JSON
std::string json;
google::protobuf::util::MessageToJsonString(pessoa, &json);
```

2. Boost Serialization (mais manual)

Código necessário:

cpp

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/serialization/string.hpp>

class MinhaClasse {
private:
    friend class boost::serialization::access;

    int id;
    std::string nome;
    std::vector<double> valores;

    // Método template de serialização (você precisa escrever isso)
    template<class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & id;
        ar & nome;
        ar & valores;
    }

public:
    // ... resto da classe
};

// Uso
#include <iostream>
#include <boost/archive/text_oarchive.hpp>

MinhaClasse obj;
obj.id = 1;
obj.nome = "Teste";

// Serializa para arquivo
std::ofstream ofs("arquivo.dat");
boost::archive::text_oarchive oa(ofs);
oa << obj;

// Deserializa
std::ifstream ifs("arquivo.dat");
boost::archive::text_iarchive ia(ifs);
MinhaClasse obj2;
ia >> obj2;
```

3. RTTR (Run Time Type Reflection) - Reflexão em tempo

de execução

Exemplo:

cpp

```
#include <rttr/registration>

struct MinhaClasse {
    int id;
    std::string nome;
    std::vector<double> valores;

    void metodo() { /* ... */ }
};

// Registro MANUAL necessário (não automático como você quer)
RTTR_REGISTRATION
{
    rttr::registration::class_<MinhaClasse>("MinhaClasse")
        .constructor<>()
        .property("id", &MinhaClasse::id)
        .property("nome", &MinhaClasse::nome)
        .property("valores", &MinhaClasse::valores)
        .method("metodo", &MinhaClasse::metodo);
}

// Uso dinâmico
void exemplo() {
    rttr::type t = rttr::type::get<MinhaClasse>();

    // Cria instância
    rttr::variant var = t.create();
    MinhaClasse& obj = var.get_value<MinhaClasse>();

    // Acessa propriedades
    rttr::property prop = t.get_property("nome");
    prop.set_value(obj, "Novo Nome");

    // Serializa para JSON (requer biblioteca adicional)
    // JSON seria gerado dinamicamente
}
```

4. Meta (Reflexão estática em headers)

Biblioteca Meta (header-only):

cpp

```
#include <meta/meta.hpp>

class Pessoa {
public:
    int idade;
    std::string nome;

    // Não precisa de registro manual!
};

// A biblioteca usa técnicas de metaprogramação template
// para extrair informações em tempo de compilação

// Definição da trait de reflexão
template<>
struct meta::register_type<Pessoa> {
    static constexpr auto name = "Pessoa";

    static constexpr auto members = meta::make_map(
        meta::member<&Pessoa::idade>("idade"),
        meta::member<&Pessoa::nome>("nome")
    );
};

// Serialização automática
void exemplo() {
    Pessoa p{25, "João"};

    auto json = meta::to_json(p);
    // json = {"idade": 25, "nome": "João"}

    Pessoa p2 = meta::from_json<Pessoa>(json);
}
```

5. C++17/20 Reflection TS (Futuro do C++)

Proposta atual (ainda não implementada):

cpp

.cpp

```
// Código hipotético - ainda não disponível
struct Pessoa {
    int id;
    std::string nome;
};

// Reflexão nativa
void exemplo() {
    using meta_info = reflexpr(Pessoa);

    // Itera sobre membros
    for_each(get_public_data_members<meta_info>(),
        [](auto member) {
            std::cout << get_name(member) << std::endl;
        });
}

// Serialização gerada automaticamente pelo compilador
std::string json = std::serialize<Pessoa>(pessoa);
}
```

6. Cereal (Similar ao Boost Serialization)

.cpp

```
#include <cereal/archives/json.hpp>
#include <cereal/types/vector.hpp>

class MinhaClasse {
public:
    int id;
    std::string nome;

private:
    // Método de serialização (ainda manual)
    template<class Archive>
    void serialize(Archive& archive) {
        archive(CEREAL_NVP(id), // NVP = Name Value Pair
                CEREAL_NVP(nome));
    }
};

// Uso
std::stringstream ss;
{
```

```
cereal::JSONOutputArchive archive(ss);
MinhaClasse obj{1, "Teste"};
archive(obj);
}
// ss.str() contém JSON
```

7. Solução Customizada (O que você está imaginando)

Seu marcador hipotético:

cpp

```
// Arquivo do usuário (Pessoa.h)
#pragma once
#include <string>
#include <vector>

SERIALIZABLE_CLASS(Pessoa) // Seu marcador
class Pessoa {
    SERIALIZABLE_FIELD(int, id)
    SERIALIZABLE_FIELD(std::string, nome)
    SERIALIZABLE_FIELD(std::vector<std::string>, emails)

public:
    // Métodos normais...
    void print() const;
};

// Seu executável gera automaticamente:
// Pessoa_serialization.h
#pragma once
#include "Pessoa.h"
#include <nlohmann/json.hpp>

template<>
struct Serializer<Pessoa> {
    static nlohmann::json to_json(const Pessoa& obj) {
        return {
            {"id", obj.id},
            {"nome", obj.nome},
            {"emails", obj.emails}
        };
    }
}
```

```

static Pessoa from_json(const nlohmann::json& j) {
    Pessoa obj;
    obj.id = j["id"];
    obj.nome = j["nome"];
    obj.emails = j["emails"].get<std::vector<std::string>>();
    return obj;
}
};

```

Comparação:

Biblioteca	Geração Automática	Overhead	Facilidade	Performance
Protobuf	✓ (via .proto)	Baixo	Média	Excelente
Boost	✗ (manual)	Alto	Baixa	Boa
RTTR	✗ (manual)	Alto	Média	Baixa
Meta	✓ (header-only)	Baixo	Alta	Excelente
Seu Tool	✓ (auto parsing)	Baixo	Alta	Excelente

Vantagens da sua abordagem:

1. Sem DSL separada (ao contrário do Protobuf)
2. Sem código manual (ao contrário do Boost/RTTR)
3. Compile-time (ao contrário do RTTR)
4. Integração direta com código existente

Desafios que você enfrentará:

cpp

```

// Exemplo de casos complexos:
template<typename T>
SERIALIZABLE_CLASS(Container) // Templates
class Container {
    std::vector<T> items;
};

```

```
SERIALIZABLE_CLASS(Derivada) // Herança
class Derivada : public Base {
    int campo_extra;
};

SERIALIZABLE_CLASS(Ciclica) // Dependência circular
class Ciclica {
    std::shared_ptr<Ciclica> outro;
};
```

Sua abordagem é promissora porque uniria o melhor de vários mundos:

- **Conveniência do Protobuf** (geração automática)
- **Performance do Meta** (compile-time)
- **Flexibilidade do Boost** (formatos múltiplos)

Quer que eu detalhe a implementação de algum aspecto específico ou mostre como lidar com casos complexos?

nesse caso eu não preciso de um marcador para os campos, eu só posso serializar texto e número não é?

Excelente ponto! Você está certo - **não precisa de marcador para cada campo** se seguir certas regras. Vou te mostrar as abordagens:

1. Abordagem Sem Marcadores (Serialização Automática de Tudo)

cpp

```
// Arquivo do usuário (Pessoa.h) - SEM marcadores nos campos!
class Pessoa {
public:
    int id; // Serializado automaticamente
    std::string nome; // Serializado automaticamente
    std::vector<std::string> emails; // Serializado automaticamente

private:
    std::string senhaHash; // Ops! Não queremos serializar isso!

protected:
    double salario; // Talvez não queira serializar também
```

};

2. Problemas da Serialização Automática Total

2.1 Campos que NÃO devem ser serializados:

cpp

```
class Usuario {
public:
    int id;
    std::string nome;

private:
    // Campos sensíveis/temporários/derivados
    std::string senhaHash; // 🔴 NÃO serializar!
    std::string sessionToken; // 🔴 NÃO serializar!
    mutable std::mutex mutex; // 🔴 IMPOSSÍVEL serializar!
    std::atomic<int> contador; // 🔴 PROBLEMÁTICO serializar!
    std::unique_ptr<Recurso> recurso; // 🔴 COMPLEXO serializar!

    // Campos derivados (cached)
    double salarioAnual; // 🔴 Talvez não queira serializar

public:
    // Ponteiros/recursos
    FILE* arquivo; // 🔴 NÃO serializar!
    std::thread thread; // 🔴 IMPOSSÍVEL serializar!

    // Callbacks/funções
    std::function<void()> callback; // 🔴 IMPOSSÍVEL serializar!
};
```

2.2 Ciclos e Ponteiros:

cpp

```
class Node {
    std::shared_ptr<Node> parent; // Ciclo!
    std::vector<std::shared_ptr<Node>> children;
};

// Serialização ingênuas causaria loop infinito ou duplicação
```

3. Soluções Possíveis para seu Tool

Opção A: Serializar Apenas Públicos (Mais Segura)

cpp

```
// Seu marcador na classe
SERIALIZABLE(Pessoa) // Apenas no nível da classe

class Pessoa {
public: // Apenas campos públicos são serializados
    int id;
    std::string nome;

private: // Campos privados ignorados
    std::string senhaHash;
    double salario;
};
```

Opção B: Marcador de Exclusão (Mais Flexível)

cpp

```
class Usuario {
    SERIALIZABLE(Usuario)

public:
    int id;
    std::string nome;
    std::string email;

    NO_SERIALIZE // Marcador para excluir campo público
    std::string sessionToken;

    std::string senhaHash; // Serializado (talvez não queira)

    NO_SERIALIZE
    mutable std::mutex mutex; // Excluído
};
```

Opção C: Serializar Tipos "Serializáveis" por Padrão

cpp

```
// Seu tool define o que é serializável por padrão:  
// - Tipos primitivos (int, float, double, bool, char)  
// - std::string, std::string_view  
// - Containers STL de tipos serializáveis  
// - Estruturas simples (POD types)  
  
// Tipos NÃO serializáveis por padrão:  
// - Ponteiros crus (int*, void*)  
// - Funções/callbacks  
// - Resources (FILE*, mutex, thread)  
// - Tipos com construtores complexos
```

4. Implementação com Heurísticas Inteligentes

cpp

```
class CppFieldAnalyzer {  
public:  
    bool shouldSerialize(const FieldInfo& field) {  
        // Regras heurísticas:  
  
        // 1. Verificar tipo  
        if (isPointerType(field.type)) return false;  
        if (isFunctionType(field.type)) return false;  
        if (isAtomicType(field.type)) return false;  
  
        // 2. Verificar modificadores  
        if (field.isMutable) return false;  
        if (field.isVolatile) return false;  
  
        // 3. Verificar nome (heurística comum)  
        static const std::vector<std::string> blacklist = {  
            "password", "senha", "token", "secret",  
            "hash", "key", "mutex", "lock", "thread"  
        };  
  
        std::string lowerName = toLower(field.name);  
        for (const auto& word : blacklist) {  
            if (lowerName.find(word) != std::string::npos) {  
                return false;  
            }  
        }  
    }  
}
```

```
// 4. Verificar se tipo tem suporte a serialização
    return isTypeSerializable(field.type);
}

private:
    bool isTypeSerializable(const TypeInfo& type) {
        // Tipos serializáveis:
        // - Fundamentais (int, float, bool, etc.)
        // - std::string
        // - std::vector<T> onde T é serializável
        // - std::map<K,V> onde K e V são serializáveis
        // - std::optional<T> onde T é serializável
        // - std::chrono::duration/time_point
        // - Enums
        // - Outras classes marcadas como SERIALIZABLE
    }
};
```

5. Exemplo Completo com Diferentes Cenários

cpp

```
// Arquivo: DataStructures.h

// Classe 1: POD simples - serialização total OK
SERIALIZABLE(Ponto3D)
class Ponto3D {
public: // Todos públicos, todos simples
    double x;
    double y;
    double z;
};

// Classe 2: Mista - precisa de cuidado
SERIALIZABLE(Usuario)
class Usuario {
public:
    int id;           // ✅ OK serializar
    std::string nome; // ✅ OK serializar

private:
    std::string senhaHash; // 🔴 Não serializar automaticamente
    double salario;       // ⚠️ Privado, mas poderia ter getter
};
```

```

public:
    // Getter para campo privado que QUEREMOS serializar
    double getSalario() const { return salario; }

    NO_SERIALIZE // ⚡ Excluir explicitamente
    std::mutex mutex;
};

// Classe 3: Complexa - alguns campos problemáticos
SERIALIZABLE(Conexao)
class Conexao {
public:
    int id;                      // ✅
    std::string endereco;        // ✅
    time_t ultimaConexao;       // ✅

    std::atomic<bool> ativa;     // ⚠️ Atomic - pode ser problemático
    std::thread worker;          // ⚡ IMPOSSÍVEL serializar
    std::function<void()> callback; // ⚡ IMPOSSÍVEL serializar

    // Solução: campos transient (não persistentes)
    TRANSIENT // ⚡ Alternativa a NO_SERIALIZE
    std::unique_ptr<Buffer> bufferTemporario;
};

```

6. Geração Inteligente de Código

cpp

```

// Seu tool gera isso:
template<>
struct Serializer<Usuario> {
    static nlohmann::json to_json(const Usuario& obj) {
        return {
            {"id", obj.id},
            {"nome", obj.nome},
            {"salario", obj.getSalario()} // Usa getter!
            // senhaHash e mutex são omitidos
        };
    }

    static Usuario from_json(const nlohmann::json& j) {
        Usuario obj;
        obj.id = j["id"];
        obj.nome = j["nome"];
        // senhaHash e mutex são omitidos
    }
}

```

```
// Se não tiver o prefixo de setter ou iterator  
// mutex é inicializado automaticamente  
return obj;  
}  
};
```

7. Recomendação para seu Tool

Sugiro esta abordagem híbrida:

cpp

```
// REGRAS DO SEU TOOL:  
  
// 1. Apenas campos públicos são considerados por padrão  
// 2. Campos privados/protegidos exigem getters/setters  
// 3. Tipos não-serializáveis são detectados automaticamente  
// 4. Marcador de exclusão para casos especiais  
// 5. Suporte a getters/setters automáticos  
  
// Exemplo final limpo:  
SERIALIZABLE(Produto)  
class Produto {  
public:  
    int id;  
    std::string nome;  
    double preco;  
  
    NO_SERIALIZE // Campo público que não quer serializar  
    static int totalProdutos;  
  
private:  
    // Privado com getter - será serializado via getter  
    int estoque;  
  
public:  
    int getEstoque() const { return estoque; }  
    void setEstoque(int e) { estoque = e; }  
};
```

8. Detecção Automática de Problemas

Seu tool poderia emitir **warnings**:

text

- ⚠ Warning: Campo 'mutex' do tipo 'std::mutex' não pode ser serializado
Classe: Usuario, Arquivo: Usuario.h:25
Sugestão: Adicione NO_SERIALIZE ou TRANSIENT
- ⚠ Warning: Campo 'callback' do tipo 'std::function' não pode ser serializado
Classe: Conexao, Arquivo: Conexao.h:42
- ✓ Info: 3 campos serão serializados automaticamente
- ✓ Info: 1 campo será serializado via getter 'getEstoque()'

Conclusão:

Você está certo que **não precisa marcar todos os campos**, mas precisa de **regras claras** para:

1. **Segurança**: Não serializar dados sensíveis
2. **Correção**: Não tentar serializar o inserializável
3. **Flexibilidade**: Permitir exclusão explícita quando necessário

A abordagem mais prática é: **serializar apenas campos públicos de tipos serializáveis**, com opção de exclusão explícita via `NO_SERIALIZE` ou similar.

Quer que eu detalhe as regras de detecção automática ou como lidar com getters/setters?

a regra seria simples: só posso serializar tipos primitivos e algumas classes, por exemplo vector, list, tuple, caso tenham como parâmetro tipo primitivo. Somente os objetos públicos devem ser serializados, se o cara colocar algo sensível é de responsabilidade dele, mas vou implementar também a macro transient pra não serializar

Excelente! Essa regra é prática e evita muitos problemas. Vou detalhar a implementação:

1. Regras de Serialização do seu Tool

cpp

```
// REGRAS DE SERIALIZAÇÃO:
// 1. ✓ Serializa apenas campos PÚBLICOS
// 2. ✓ Tipos primitivos: int, float, double, bool, char, etc.
// 3. ✓ std::string, std::string_view
// 4. ✓ Containers STL de tipos serializáveis:
//     - std::vector<T>, std::list<T>, std::deque<T>
//     - std::array<T, N>, std::set<T>, std::unordered_set<T>
//     - std::map<K,V>, std::unordered_map<K,V>
//     - std::pair<T1,T2>, std::tuple<T...>
// 5. ✓ std::optional<T>, std::variant<T...>
// 6. ✓ std::chrono::duration, std::chrono::time_point
// 7. ✓ Enums (enum, enum class)
// 8. ✗ Ponteiros crus: T*, void*
// 9. ✗ Smart pointers: std::unique_ptr, std::shared_ptr
// 10. ✗ Tipos complexos: std::function, std::thread, std::mutex
// 11. ✗ Arrays C-style: T[]
// 12. ✓ Outras classes marcadas com SERIALIZABLE
```

2. Macros para o Usuário

cpp

```
// Macros.h (seu header que os usuários incluem)
#pragma once

#define SERIALIZABLE(Classname) \
/* Marca a classe para o seu tool encontrar */

#define TRANSIENT \
/* Marca campo para não serializar */ \
/* Pode ser implementado com atributo [[maybe_unused]] */ \
[[maybe_unused]]
```

3. Implementação do Parser com essas Regras

cpp

```
class TypeChecker {
public:
    bool isSerializableType(const std::string& typeName) {
        // Tipos primitivos
        static const std::set<std::string> primitiveTypes = {
            "int", "int8_t", "int16_t", "int32_t", "int64_t",
            "uint8_t", "uint16_t", "uint32_t", "uint64_t",
```

```
"float", "double", "long double",
"bool", "char", "wchar_t", "char16_t", "char32_t",
"size_t", "ssize_t", "ptrdiff_t"
};

if (primitiveTypes.count(typeName)) return true;

// Strings
if (typeName == "std::string") return true;
if (typeName == "std::string_view") return true;

// Verificar containers
return isSerializableContainer(typeName);
}

bool isSerializableContainer(const std::string& typeName) {
    // Padrões de containers STL
    // Exemplos: std::vector<int>, std::map<std::string, double>

    // Extrair template arguments
    auto [baseType, templateArgs] = extractTemplateInfo(typeName);

    // Containers permitidos
    static const std::set<std::string> allowedContainers = {
        "std::vector", "std::list", "std::deque", "std::array",
        "std::set", "std::unordered_set", "std::multiset",
        "std::map", "std::unordered_map", "std::multimap",
        "std::pair", "std::tuple", "std::optional", "std::variant"
    };

    if (allowedContainers.count(baseType)) {
        // Verificar se todos os template args são serializáveis
        return areAllTemplateArgsSerializable(templateArgs);
    }

    return false;
}
};
```

4. Exemplos de Código Válido/Inválido

cpp

```
// Arquivo do usuário: Exemplo.h
#include "Macros.h"
```

```
// ✅ CLASSE VÁLIDA 1: Tipos básicos
SERIALIZABLE(Pessoa)
class Pessoa {
public:
    int id;
    std::string nome;
    double altura;
    bool ativo;
    std::vector<std::string> habilidades;
    std::map<std::string, int> configuracoes;
};

// ✅ CLASSE VÁLIDA 2: Com transient
SERIALIZABLE(Usuario)
class Usuario {
public:
    int id;
    std::string username;

    TRANSIENT // Não será serializado
    std::string sessionToken;

    std::vector<int> permissoes;
};

// ✅ CLASSE VÁLIDA 3: Containers aninhados
SERIALIZABLE(Relatorio)
class Relatorio {
public:
    std::string titulo;
    std::vector<std::map<std::string, double>> dados;
    std::tuple<int, std::string, bool> metadados;
};

// ❌ CLASSE PROBLEMÁTICA 1: Ponteiros
SERIALIZABLE(EstruturaInvalida)
class EstruturaInvalida {
public:
    int* ponteiro; // ❌ Ponteiro cru - será ignorado
    std::unique_ptr<int> smartPtr; // ❌ Smart pointer - será ignorado
    std::function<void()> callback; // ❌ std::function - será ignorado
};

// ❌ CLASSE PROBLEMÁTICA 2: Campos privados
SERIALIZABLE(ClassePrivada)
class ClassePrivada {
```

```

private: // Campos privados são IGNORADOS pelo seu tool
    int secreto; // ❌ Não será serializado (e nem deve!)

public:
    int publico; // ✅ Será serializado
};


```

5. Geração de Código com Transient

cpp

```

// Seu tool gera isso:
template<>
struct Serializer<Usuario> {
    static nlohmann::json to_json(const Usuario& obj) {
        nlohmann::json j;
        j["id"] = obj.id; // ✅ Serializa
        j["username"] = obj.username; // ✅ Serializa
        // sessionToken é TRANSIENT - omitido ❌
        j["permissoes"] = obj.permissoes; // ✅ Serializa
        return j;
    }

    static Usuario from_json(const nlohmann::json& j) {
        Usuario obj;
        obj.id = j["id"];
        obj.username = j["username"];
        // sessionToken não é desserializado
        // Mantém valor default ou precisa ser setado depois
        obj.permissoes = j["permissoes"].get<std::vector<int>>();
        return obj;
    }
};

```

6. Warnings/Errors do seu Tool

cpp

```

class CodeAnalyzer {
public:
    void analyzeClass(const ClassInfo& cls) {
        std::cout << "Analizando classe: " << cls.name << "\n";
        for (const auto& field : cls.fields) {

```

```
        if (!field.isPublic) {
            // Apenas log, não é erro
            std::cout << "  Info: Campo privado '" << field.name
                << "' será ignorado\n";
            continue;
        }

        if (field.hasTransientMacro) {
            std::cout << "  Info: Campo TRANSIENT '" << field.name
                << "' será omitido\n";
            continue;
        }

        if (!typeChecker.isSerializableType(field.type)) {
            // WARNING, não ERROR - o campo é simplesmente ignorado
            std::cout << "  ! Warning: Tipo não serializável '"
                << field.type << "' no campo '"
                << field.name << "'\n";
            std::cout << "      O campo será omitido da serialização\n";
        }
    }
};
```

7. Output do Tool (Exemplo)

text

```
$ cpp-serializer-gen --input Usuario.h
```

- Analizando: Usuario.h
- Classe 'Usuario' marcada como SERIALIZABLE
- Campos encontrados:
 - id (int) ✓ Serializável
 - username (std::string) ✓ Serializável
 - sessionToken (std::string) ! TRANSIENT - Omitido
 - permissoes (std::vector<int>) ✓ Serializável

✨ Gerando: Usuario_serialization.h
✨ Gerando: Usuario_serialization.cpp

8. Código Gerado Final

cpp

```
// Usuario_serialization.h (gerado automaticamente)
#pragma once
#include "Usuario.h"
#include <nlohmann/json.hpp>
#include <vector>
#include <map>
#include <string>

// Serialização para JSON
template<>
struct JsonSerializer<Usuario> {
    static void to_json(nlohmann::json& j, const Usuario& obj) {
        j = nlohmann::json{
            {"id", obj.id},
            {"username", obj.username},
            {"permissoes", obj.permissoes}
            // sessionToken omitido (TRANSIENT)
            // mutex omitido (tipo não serializável)
        };
    }

    static void from_json(const nlohmann::json& j, Usuario& obj) {
        obj.id = j.at("id").get<int>();
        obj.username = j.at("username").get<std::string>();
        obj.permissoes = j.at("permissoes").get<std::vector<int>>();
        // sessionToken mantém valor default
        // mutex inicializado automaticamente
    }
};

// Também pode gerar outros formatos
template<>
struct BinarySerializer<Usuario> { /* ... */ };


```

9. Vantagens dessa Abordagem

1. **Simplicidade:** Regras claras e previsíveis
2. **Segurança:** Campos privados automaticamente protegidos
3. **Flexibilidade:** TRANSIENT para casos excepcionais
4. **Robustez:** Tipos complexos são silenciosamente ignorados

5. **Performance:** Só gera código para o que é realmente serializável

10. Extensibilidade Futura

Se depois quiser adicionar suporte a tipos customizados:

cpp

```
// Versão futura poderia suportar:  
SERIALIZABLE_WITH_CUSTOM(MinhaClasse)  
class MinhaClasse {  
public:  
    std::string nome;  
  
    CUSTOM_SERIALIZE(MeuTipoComplexo) objComplexo;  
    // O usuário forneceria serializador customizado  
};
```

Essa abordagem é pragmática e resolve 90% dos casos de uso comuns. O usuário que precisa serializar ponteiros ou tipos complexos terá que implementar manualmente ou refatorar o código - o que é uma boa prática de qualquer forma!

Quer que eu detalhe alguma parte específica, como a detecção de tipos STL ou a geração do código CMake?

vou começar o projeto agr qual a melhor versão do c++ pra isso? a mais nova?

Excelente que vai começar! Vou te dar uma análise detalhada:

🎯 Recomendação: C++17 é o sweet spot

Para um tool como o seu, C++17 é a melhor escolha. Eis por que:



Comparação das Versões

Versão	Status para seu Tool	Vantagens	Desvantagens
C++11	✓ Estável mas limitado	Máxima compatibilidade	Falta features modernas
C++14	✓ Bom equilíbrio	Melhor que C++11	Ainda não tem tudo

C++17	RECOMENDADO	Features maduras	Compatibilidade n.
C++20	Futuro próximo	Muito poderoso	Supporte do compilador em evolução
C++23	x Muito novo	Features inovadoras	Compiladores ainda suportam tudo

🔥 Por que C++17 é perfeito para seu tool:

1. Features que você VAI usar:

cpp

```
// Structured bindings (perfeito para parsing)
auto [baseType, templateArgs] = extractTemplateInfo(typeStr);

// std::optional (para resultados que podem falhar)
std::optional<ClassInfo> parseClass(const std::string& code);

// std::variant (para AST nodes)
using Node = std::variant<ClassNode, StructNode, EnumNode>;

// std::string_view (performance no parsing)
void processCode(std::string_view code);

// Filesystem (percorrer projetos)
for (const auto& entry : fs::directory_iterator(path)) {
    if (entry.path().extension() == ".h") {
        processFile(entry.path());
    }
}

// if constexpr (metaprogramação)
template<typename T>
if constexpr (std::is_same_v<T, std::string>) {
    // Tratamento especial para strings
}
```

2. Exemplo de código do seu parser usando C++17:

cpp

```

#include <filesystem>
#include <optional>
#include <string_view>
#include <variant>

namespace fs = std::filesystem;

class CppParser {
public:
    struct ParsedClass {
        std::string name;
        std::vector<Field> fields;
        std::vector<Method> methods;
        fs::path sourceFile;
    };
};

std::optional<ParsedClass> parseHeader(const fs::path& file) {
    auto content = readFile(file);
    if (!content) return std::nullopt;

    return parseClassDeclaration(std::string_view(*content));
}

private:
    std::optional<std::string> readFile(const fs::path& path) {
        if (!fs::exists(path)) return std::nullopt;

        std::ifstream file(path);
        if (!file) return std::nullopt;

        return std::string(
            std::istreambuf_iterator<char>(file),
            std::istreambuf_iterator<char>()
        );
    }
};

```

3. CMake para C++17:

```

cmake

# CMakeLists.txt do SEU tool
cmake_minimum_required(VERSION 3.10)
project(cpp-serializer-gen VERSION 1.0.0)

```

```
# Exige C++17
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

# Encontra filesystem (necessário no GCC < 9)
find_package(Filesystem REQUIRED)

add_executable(cpp-serializer-gen
    src/main.cpp
    src/Parser.cpp
    src/CodeGenerator.cpp
    src/CMakeAnalyzer.cpp
)

target_link_libraries(cpp-serializer-gen
    PRIVATE std::filesystem
)

# Para Clang/LLVM se for usar libtooling
find_package(LLVM REQUIRED CONFIG)
find_package(Clang REQUIRED CONFIG)
```

🚀 C++20: Vale a pena considerar?

Features úteis do C++20:

cpp

```
// Concepts (perfeito para seu type checker)
template<typename T>
concept Serializable = requires(T t) {
    { Serializer<T>::to_json(t) } -> std::same_as<nlohmann::json>;
    { Serializer<T>::from_json(nlohmann::json{}) } -> std::same_as<T>;
};

// Ranges (para processamento de código)
auto serializableClasses = sourceFiles
    | std::views::filter(isCppFile)
    | std::views::transform(parseFile)
    | std::views::filter(hasSerializableMacro)
    | std::ranges::to<std::vector>();

// std::format (para geração de código)
```

```
std::string generateSerializer(const ClassInfo& cls) {
    return std::format(R"(

template<>
struct Serializer<{}> {{
    static nlohmann::json to_json(const {}& obj) {{
        return {{

    }}
    }};;
}}
```

}", cls.name, cls.name, generateFieldsJson(cls.fields));
}

Problemas do C++20 hoje:

- **GCC**: Suporte completo a partir da versão 11 (2021)
- **Clang**: Suporte completo a partir da versão 13 (2021)
- **MSVC**: Suporte razoável, mas alguns bugs
- **Distribuições Linux**: Ubuntu 20.04 tem GCC 9, precisa PPAs

Recomendações Práticas:

Para DESENVOLVIMENTO (seu tool):

```
cmake

# Use C++17 para estabilidade máxima
# Se quiser arriscar, C++20 com fallback:
if (CMAKE_CXX_COMPILER_ID STREQUAL "GNU" AND
    CMAKE_CXX_COMPILER_VERSION VERSION_GREATER_EQUAL 11)
    set(CMAKE_CXX_STANDARD 20)
else()
    set(CMAKE_CXX_STANDARD 17)
endif()
```

Para USO (dos usuários do seu tool):

```
cpp

// Seu tool deve ser flexível com versões C++
// Usuários podem usar C++11 14 17 20
```

```
// Usuários podem usar C++11, 14, 17, 20...
// O código GERADO pelo seu tool:
#ifndef __cplusplus >= 201703L
    // Usa C++17 features se disponível
    #include <optional>
    #include <string_view>
#else
    // Fallback para C++11/14
    // Implementações próprias ou boost
#endif
```

Dependências Recomendadas:

cmake

```
# Para C++17 (todas as distros recentes têm)
find_package(Filesystém REQUIRED) # Pode precisar de -lstdc++fs no GCC < 9

# Bibliotecas recomendadas para seu tool:
# 1. nlohmann/json (header-only) - para exemplos/demo
# 2. CLI11 (header-only) - para parsing de linha de comando
# 3. fmt (opcional) - formatação melhor que std::format do C++20
# 4. Boost (opcional) - se precisar de fallback para C++11

# Se for usar Clang AST:
find_package(LLVM REQUIRED CONFIG)
find_package(Clang REQUIRED CONFIG)
```

Estrutura Inicial do Projeto:

bash

```
cpp-serializer-gen/
├── CMakeLists.txt          # C++17 requerido
└── src/
    ├── main.cpp            # Ponto de entrada
    └── Parser/
        ├── CppParser.cpp   # Usa C++17 features
        └── TypeChecker.cpp
    ├── Generator/         # Geração de código
    └── Utils/              # Helpers
    └── include/            # Headers públicos
    └── tools/               # Scripts auxiliares
```

```
└─ tests/          # Testes
```

🎯 Veredito Final:

Comece com C++17 porque:

1. Suporte universal (GCC 7+, Clang 5+, MSVC 2017+)
2. Features maduras e bem testadas
3. Não precisa de workarounds
4. Todas distros Linux recentes suportam
5. Bom equilíbrio entre modernidade e compatibilidade

Se depois de 6 meses quiser migrar para C++20:

1. As features estarão mais estáveis
2. Você já terá o core funcionando
3. Poderá refatorar partes específicas
4. Mais usuários terão compiladores compatíveis



Primeiros Passos:

bash

```
# 1. Crie o projeto
mkdir cpp-serializer-gen
cd cpp-serializer-gen

# 2. CMakeLists.txt inicial
cat > CMakeLists.txt << 'EOF'
cmake_minimum_required(VERSION 3.10)
project(cpp-serializer-gen VERSION 0.1.0)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(cpp-serializer-gen src/main.cpp)

# Para GCC < 9 pode precisar:
target_link_libraries(cpp-serializer-gen stdc++fs)
EOF
```

```
# 3. Estrutura básica  
mkdir -p src include tests
```

Quer que eu te mostre o código inicial do parser ou da geração de código?

vou fazer com 20. ja temos c++ 26. deixa eu te perguntar no projeto que vai usar essa tool o cmake poderia executar a tool antes de compilar?

Sim, absolutamente! Isso é exatamente a forma correta de integrar seu tool. Vou te mostrar várias estratégias de integração com CMake:

Estratégia 1: Pré-compilação Automática (Recomendada)

cmake

```
# No CMakeLists.txt do projeto USUÁRIO
cmake_minimum_required(VERSION 3.15) # Para add_custom_command com BYPRODUCTS
project(MeuProjeto VERSION 1.0.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# 1. Encontra seu tool (instalado no sistema)
find_program(CPP_SERIALIZER_GEN cpp-serializer-gen REQUIRED)

# 2. Lista de arquivos que têm classes serializáveis
set(SERIALIZABLE_HEADERS
    src/Usuario.h
    src/Produto.h
    src/Pedido.h
)
# 3. Lista de arquivos gerados (output do seu tool)
set(GENERATED_SERIALIZATION
    ${CMAKE_CURRENT_BINARY_DIR}/generated/Usuario_serialization.h
    ${CMAKE_CURRENT_BINARY_DIR}/generated/Produto_serialization.h
    ${CMAKE_CURRENT_BINARY_DIR}/generated/Pedido_serialization.h
)
# 4. Comando customizado para gerar os serializadores
add_custom_command(
```

```

OUTPUT ${GENERATED_SERIALIZATION}
COMMAND ${CPP_SERIALIZER_GEN}
ARGS --input ${SERIALIZABLE_HEADERS}
    --output-dir ${CMAKE_CURRENT_BINARY_DIR}/generated
    --format json,binary
DEPENDS ${SERIALIZABLE_HEADERS}
COMMENT "Gerando serializadores C++..."
WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
BYPRODUCTS ${GENERATED_SERIALIZATION}
)

# 5. Biblioteca/executável principal
add_library(meu_projeto_lib
    src/Usuario.cpp
    src/Produto.cpp
    src/Pedido.cpp
    # Inclui os headers gerados como source files
    ${GENERATED_SERIALIZATION}
)

```

6. Inclui diretório dos arquivos gerados

```

target_include_directories(meu_projeto_lib PUBLIC
    ${CMAKE_CURRENT_BINARY_DIR}/generated
)

```

🎯 Estratégia 2: Detecção Automática de Arquivos

cmake

```

# CMakeLists.txt mais inteligente
find_program(CPP_SERIALIZER_GEN cpp-serializer-gen REQUIRED)

# Função para encontrar arquivos com SERIALIZABLE
function(find_serializable_files DIR OUTPUT_VAR)
    file(GLOB_RECURSE HEADER_FILES "${DIR}/*.h" "${DIR}/*.hpp")
    set(SERIALIZABLE_FILES)

    foreach(HEADER ${HEADER_FILES})
        file(READ "${HEADER}" HEADER_CONTENT)
        if(HEADER_CONTENT MATCHES "SERIALIZABLE\\\"")
            list(APPEND SERIALIZABLE_FILES ${HEADER})
            message(STATUS "Encontrado serializable em: ${HEADER}")
        endif()
    endforeach()

    set(${OUTPUT_VAR} ${SERIALIZABLE_FILES})
endfunction()

```

© 2024 DeepSeek. Todos os direitos reservados.

```

SCALAR_VARIABLE_SERIALIZABLE_FILES_HANDLER()
endfunction()

# Encontra automaticamente
find_serializable_files(${CMAKE_CURRENT_SOURCE_DIR}/src SERIALIZABLE_HEADERS)

# Gera comandos dinamicamente
foreach(HIGHLIGHTED_HEADER ${SERIALIZABLE_HEADERS})
    get_filename_component(BASE_NAME ${HEADER} NAME_WE)
    set(GEN_FILE ${CMAKE_CURRENT_BINARY_DIR}/generated/${BASE_NAME}_serializer
n.h)

    add_custom_command(
        OUTPUT ${GEN_FILE}
        COMMAND ${CPP_SERIALIZER_GEN}
        ARGS --input ${HEADER} --output ${GEN_FILE}
        DEPENDS ${HEADER}
        COMMENT "Gerando serializador para ${BASE_NAME}"
    )

    list(APPEND GENERATED_FILES ${GEN_FILE})
endforeach()

# Adiciona ao target
add_executable(meu_app src/main.cpp ${GENERATED_FILES})
target_include_directories(meu_app PUBLIC ${CMAKE_CURRENT_BINARY_DIR}/generated)

```

🎯 Estratégia 3: Como Submódulo do Git

bash

```
# Usuário adiciona seu tool como submódulo
git submodule add https://github.com/seu-user/cpp-serializer-gen
```

cmake

```

# CMakeLists.txt do projeto usuário
# Opção A: Build seu tool junto com o projeto
add_subdirectory(cpp-serializer-gen)
# Agora CPP_SERIALIZER_GEN é o target do executável

# Opção B: Se seu tool tiver INSTALL no próprio CMake
find_package(cpp-serializer-gen CONFIG REQUIRED)
# Ou
find_program(CPP_SERIALIZER_GEN cpp-serializer-gen REQUIRED)
```

🎯 Estratégia 4: Com Configuração por Arquivo

```
cmake

# .serializer-config.cmake (na raiz do projeto usuário)
set(SERIALIZER_CONFIG
    FORMATS "json;binary;xml"
    OUTPUT_DIR "${CMAKE_BINARY_DIR}/generated_serializers"
    INCLUDE_PRIVATE_FIELDS OFF
    GENERATE_GETTERS_SETTERS ON
)

# No CMakeLists.txt principal
configure_file(
    .serializer-config.cmake
    ${CMAKE_BINARY_DIR}/serializer-config.cmake
)

add_custom_command(
    OUTPUT ${GENERATED_FILES}
    COMMAND ${CPP_SERIALIZER_GEN}
    ARGS --config ${CMAKE_BINARY_DIR}/serializer-config.cmake
        --project ${CMAKE_CURRENT_SOURCE_DIR}
    DEPENDS ${ALL_SOURCE_FILES}
    COMMENT "Gerando serializadores..."
)
```

🎯 Estratégia 5: Para Projetos Grandes (CMake 3.18+)

```
cmake

# Usa filesets para organização
add_library(meu_projeto

target_sources(meu_projeto
PRIVATE
    FILE_SET serializable_headers TYPE HEADERS
    BASE_DIRS ${CMAKE_CURRENT_SOURCE_DIR}/src
    FILES
        src/Usuario.h
        src/Produto.h
PRIVATE
    FILE_SET generated_serializers TYPE HEADERS
```

```
BASE_DIRS ${CMAKE_CURRENT_BINARY_DIR}/generated
FILES
${GENERATED_SERIALIZATION}
)

# Geração com dependências em file_sets
add_custom_command(
    OUTPUT ${GENERATED_SERIALIZATION}
    COMMAND ${CPP_SERIALIZER_GEN}
    ARGS --file-set serializable_headers
        --output-file-set generated_serializers
    DEPENDS ${<TARGET_PROPERTY:meu_projeto,HEADERS>}
    COMMENT "Gerando serializadores..."
)
```

🔧 Implementação do seu Tool (C++20):

cpp

```
// main.cpp do seu tool
#include <filesystem>
#include <iostream>
#include <ranges>
#include <format>

namespace fs = std::filesystem;

int main(int argc, char* argv[]) {
    // Parse arguments
    std::vector<std::string> args(argv + 1, argv + argc);

    // Exemplo: cpp-serializer-gen --cmake CMakeLists.txt
    if (auto it = std::ranges::find(args, "--cmake"); it != args.end()) {
        // Modo integração CMake
        fs::path cmakeFile = *it;
        auto project = parseCMakeProject(cmakeFile);

        for (const auto& sourceFile : project.sourceFiles
            | std::views::filter(isHeaderFile)) {

            if (containsSerializable(sourceFile)) {
                auto serializationCode = generateSerializer(sourceFile);
                writeGeneratedFile(serializationCode, getOutputPath(sourceFile));
            }
        }
    }
}
```

```
    }

    return 0;
}
```



Exemplo de Projeto Usuário Completo:

bash

```
projeto-usuario/
├── CMakeLists.txt          # Usa seu tool
├── src/
│   ├── main.cpp
│   ├── Usuario.h           # Tem SERIALIZABLE
│   ├── Usuario.cpp
│   ├── Produto.h           # Tem SERIALIZABLE
│   └── Produto.cpp
└── build/
    ├── generated/          # Arquivos gerados pelo seu tool
    │   ├── Usuario_serialization.h
    │   └── Produto_serialization.h
    └── meu_app               # Executável final
└── .serializer-config.yaml # Config do seu tool (opcional)
```

cmake

```
# CMakeLists.txt do usuário (C++20 também! )
cmake_minimum_required(VERSION 3.20)
project(MeuApp VERSION 1.0.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# INSTALAÇÃO do seu tool (uma vez)
# find_package(cpp-serializer-gen REQUIRED)
# OU
# Baixa automaticamente com FetchContent

include(FetchContent)
FetchContent_Declare(
    cpp_serializer_gen
    GIT_REPOSITORY https://github.com/seu-user/cpp-serializer-gen
    GIT_TAG main
)
FetchContent_MakeAvailable(cpp_serializer_gen)
```

```

# Agora usa o tool
add_custom_target(generate_serializers ALL
    COMMAND $<TARGET_FILE:cpp-serializer-gen>
        --project ${CMAKE_CURRENT_SOURCE_DIR}
        --output ${CMAKE_CURRENT_BINARY_DIR}/generated
    COMMENT "Executando cpp-serializer-gen..."
    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
)

add_executable(meu_app src/main.cpp)
add_dependencies(meu_app generate_serializers)
target_include_directories(meu_app PUBLIC
    ${CMAKE_CURRENT_BINARY_DIR}/generated
)

```

Dica de Performance:

cmake

```

# Geração incremental - só regenera o que mudou
add_custom_command(
    OUTPUT ${GENERATED_FILES}
    COMMAND ${CPP_SERIALIZER_GEN}
    ARGS --incremental
        --cache ${CMAKE_CURRENT_BINARY_DIR}/.serializer-cache
        ${SERIALIZABLE_HEADERS}
    DEPENDS ${SERIALIZABLE_HEADERS}
    COMMENT "Gerando serializadores (incremental)..."
)

```

Considerações Importantes:

1. **Timing:** O tool roda na fase de `cmake --build`, não no `configure`
2. **Dependências:** Se um header mudar, CMake regenera automaticamente
3. **Cache:** Use `CMAKE_CURRENT_BINARY_DIR` para outputs
4. **Portabilidade:** Use `cmake -E` para comandos cross-platform
5. **IDEs:** Arquivos gerados aparecem no IDE se estiverem no `target_sources`

Vantagens dessa Abordagem:

- **Transparente:** Usuário só precisa marcar classes com `SERIALIZABLE`
- **Automático:** Roda antes de cada build se necessário
- **Integrado:** Funciona com make, ninja, Visual Studio, Xcode
- **Cacheável:** CMake gerencia dependências automaticamente
- **Cross-platform:** Mesmo comando funciona em todos OS

Resposta curta: SIM, o CMake pode (e deve!) executar seu tool antes de compilar. É o padrão para code generators em C++ (Protobuf, FlatBuffers, thrift, etc).

Quer que eu detalhe a implementação do modo `--cmake` do seu tool ou como parsear o `CMakeLists.txt`?

outra pergunta, como eu posso integrar a ide pra quando o cara marcar como serializable ele ja ter as funções serialize e deserialize

Excelente pergunta! Isso é **code generation integrada à IDE** - um diferencial enorme. Vou te mostrar várias estratégias:

🎯 1. Language Server Protocol (LSP) - Método Moderno

cpp

```
// lsp_server.cpp no seu tool
#include <nlohmann/json.hpp>
#include <optional>
#include <filesystem>

class SerializerLSPServer {
public:
    void onDidChangeTextDocument(const nlohmann::json& params) {
        auto text = params["contentChanges"][0]["text"];
        auto uri = params["textDocument"]["uri"];

        // Detecta quando usuário digita SERIALIZABLE
        if (text.contains("SERIALIZABLE")) {
            // Analisa a classe
            auto classInfo = parseClass(text);

            // Gera código de serialização
            auto serializationCode = generateSerialization(classInfo);
        }
    }
}
```

```

        // Envia para IDE inserir o código
        sendCodeAction(uri, serializationCode);
    }

}

private:
    void sendCodeAction(const std::string& uri,
                        const std::string& code) {
        nlohmann::json response = {
            {"jsonrpc", "2.0"},
            {"method", "textDocument/codeAction"},
            {"params", {
                {"textDocument", [{"uri", uri}]},
                {"range", {/* posição */}},
                {"context", {
                    {"diagnostics", {}},
                    {"only", {"quickfix"}}
                }}
            }},
            {"result", {{
                {"title", "Generate serialize/deserialize methods"},
                {"kind", "quickfix"},
                {"edit", {
                    {"changes", {
                        {uri, {{
                            {"range", /* posição */},
                            {"newText", code}
                        }}}
                    }}
                }}
            }}}
        };
    }
};

```

🎯 2. Clangd + Compilation Database (Mais Simples)

Seu tool gera um `compile_commands.json` customizado:

```

json

// .clangd no projeto do usuário
CompileFlags:
    Add:
        - "-Xclang"

```

```

  clang
  - "-plugin"
  - "-Xclang"
  - "cpp-serializer-gen"
  - "-Xclang"
  - "-plugin-arg-cpp-serializer-gen"
  - "-Xclang"
  - "generate-inline"
}

cpp

// Plugin Clang para seu tool
class SerializerASTConsumer : public clang::ASTConsumer {
public:
    void HandleTranslationUnit(clang::ASTContext &ctx) override {
        auto decls = ctx.getTranslationUnitDecl()->decls();

        for (auto* decl : decls) {
            if (auto* record = llvm::dyn_cast<clang::CXXRecordDecl>(decl)) {
                // Verifica se tem SERIALIZABLE
                if (hasSerializableAttribute(record)) {
                    // Gera métodos inline
                    auto methods = generateInlineMethods(record);

                    // Cria "fix-it" hint para IDE
                    auto diag = ctx.getDiagnostics().report(
                        record->getLocation(),
                        clang::diag::remark_serializer_available
                    );
                    diag << methods;
                }
            }
        }
    }
};

};


```

3. VS Code Extension (Mais Popular)

```

json

// package.json da sua extension
{
  "name": "cpp-serializer-helper",
  "publisher": "seu-nome",
  "activationEvents": ["onLanguage:cpp"],

```

```
"contributes": {  
    "commands": [{  
        "command": "cpp-serializer.generate",  
        "title": "Generate Serialization Methods"  
    }],  
    "menus": {  
        "editor/context": [{  
            "command": "cpp-serializer.generate",  
            "when": "editorHasSelection && editorLangId == cpp"  
        }]  
    }  
}
```

typescript

```
// extension.ts

import * as vscode from 'vscode';
import { exec } from 'child_process';

export function activate(context: vscode.ExtensionContext) {
    // Comando para gerar métodos
    let disposable = vscode.commands.registerCommand(
        'cpp-serializer.generate',
        async () => {
            const editor = vscode.window.activeTextEditor;
            if (!editor) return;

            const text = editor.document.getText();
            const position = editor.selection.active;

            // Chama seu tool via CLI
            exec(`cpp-serializer-gen --inplace "${editor.document.fileName}"`,
                (error, stdout) => {
                    if (!error) {
                        // Insere código gerado
                        editor.edit(editBuilder => {
                            const snippet = new vscode.SnippetString(stdout);
                            editBuilder.insert(position, snippet);
                        });
                    }
                });
        });
}

// Code Lens (botão acima da classe)
const codeLensProvider = {
```

```

    provideCodeLenses(document) {
        const text = document.getText();
        const matches = text.match(/SERIALIZABLE\((\w+)\)/g);

        return matches.map(match => {
            const line = document.positionAt(text.indexOf(match)).line;
            return new vscode.CodeLens(
                new vscode.Range(line, 0, line, 0),
                {
                    title: "✨ Generate Serialization",
                    command: "cpp-serializer.generate",
                    arguments: [document.uri, line]
                }
            );
        });
    });

    context.subscriptions.push(
        vscode.languages.registerCodeLensProvider('cpp', codeLensProvider),
        disposable
    );
}

```

🎯 4. CLion/IntelliJ Plugin (Para JetBrains IDEs)

kotlin

```

// CppSerializerAction.kt
class CppSerializerAction : AnAction() {
    override fun actionPerformed(e: AnActionEvent) {
        val editor = e.getData(CommonDataKeys.EDITOR) ?: return
        val psiFile = e.getData(CommonDataKeys.PSI_FILE) as? CppFile ?: return

        // Encontra classes com SERIALIZABLE
        psiFile.accept(object : CppRecursiveElementVisitor() {
            override fun visitClass(o: CppClass) {
                if (o.textContains("SERIALIZABLE")) {
                    // Gera métodos usando seu tool
                    val methods = generateMethods(o)

                    // Insere no código
                    WriteCommandAction.runWriteCommandAction(project) {
                        o.addAfter(methods, o.lastChild)
                    }
                }
            }
        })
    }
}

```

```
        }

    }

}

override fun update(e: AnActionEvent) {
    // Mostra ação só quando cursor está em classe SERIALIZABLE
    val psiFile = e.getData(CommonDataKeys.PSI_FILE)
    e.presentation.isEnabled = psiFile?.text?.contains("SERIALIZABLE") ?: false
    if (psiFile?.text?.contains("SERIALIZABLE") == true) {
        e.presentation.isEnabled = true
    }
}
```

🎯 5. QuickFix via Clang Diagnostics (Funciona em várias IDEs)

cpp

```
// No seu tool, quando analisa código:
void generateQuickFix(const ClassInfo& cls) {
    std::cout << R"(

#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wunknown-pragmas"
#pragma clang diagnostic ignored "-Wpragma-once-outside-header"

#pragma clang diagnostic error "SERIALIZABLE_CLASS_DETECTED"
// ^ Isso faz a IDE mostrar um "erro" clicável

/*
QUICKFIX: Generate serialization methods for )" << cls.name << R"(

Click here or press Alt+Enter to generate:

template<typename Archive>
void serialize(Archive& ar) {
)" << generateFieldsSerialization(cls.fields) << R"(

}
*/



#pragma clang diagnostic pop
)";
}
```

🎯 6. Template Snippets (Mais Simples)

json

```
// .vscode/cpp.code-snippets
{
  "SERIALIZABLE": {
    "prefix": "SERIALIZABLE",
    "body": [
      "SERIALIZABLE(${1:ClassName})",
      "class ${1:ClassName} {",
      "public:",
      "    // Place cursor here and press Ctrl+Space for serialization",
      "    $0",
      "}";
    ],
    "description": "Create a serializable class"
  },
  "Generate Serialization": {
    "prefix": "genser",
    "body": [
      "template<typename Archive>",
      "void serialize(Archive& ar) {",
      "    ar & ${1:field1};",
      "    ar & ${2:field2};",
      "}",
      "",
      "template<typename Archive>",
      "void deserialize(Archive& ar) {",
      "    ar & ${1:field1};",
      "    ar & ${2:field2};",
      "}"
    ]
  }
}
```

🎯 7. Implementação Híbrida (Recomendada)

Crie uma **DAP Server** (Debug Adapter Protocol) que também faz code generation:

python

```
# dap_server.py - Um servidor simples em Python
import json
import subprocess
import sys
```

```

class SerializerDAPServer:
    def handle_request(self, request):
        if request['command'] == 'generateSerialization':
            # Chama seu tool C++
            result = subprocess.run(
                ['cpp-serializer-gen', '--json', request['source']],
                capture_output=True, text=True
            )

            # Retorna edições para IDE aplicar
            return {
                'success': True,
                'edits': json.loads(result.stdout)
            }

# .vscode/launch.json
{
    "type": "cpp-serializer",
    "request": "launch",
    "name": "Generate Serialization",
    "program": "${workspaceFolder}/.tools/cpp-serializer-dap"
}

```

🎯 8. Como Funcionaria na Prática

No arquivo do usuário:

```

cpp

// Usuario.h - O usuário digita:
SERIALIZABLE(Usuario)
class Usuario {
public:
    int id;
    std::string nome;
    // ↑ IDE mostra lâmpada 💡 ou Code Lens
};
```

Usuário clica na lâmpada ou pressiona Ctrl+. → Seleciona "Generate serialization methods"

IDE automaticamente insere:

cpp

```
SERIALIZABLE(Usuario)
class Usuario {
public:
    int id;
    std::string nome;

    // ===== GERADO AUTOMATICAMENTE =====
    template<typename Archive>
    void serialize(Archive& ar) {
        ar & id;
        ar & nome;
    }

    template<typename Archive>
    void deserialize(Archive& ar) {
        ar & id;
        ar & nome;
    }
    // =====
};
```

🎯 9. Setup Fácil para Usuários

bash

```
# Instalação em 1 comando
pip install cpp-serializer-ide # Ou via npm/vsix

# Config automática
cpp-serializer-ide --setup
# Isso cria:
# - .vscode/extensions.json
# - .vscode/settings.json
# - .clangd
# - .idea/cppSerializer.xml (CLion)
```



Comparação das Abordagens:

Método	Dificuldade	Compatibilidade	Experiência
LSP Server	Alta	Todas IDEs	Excelente

Clang Plugin	Média	Clang-based	Muito boa
VS Code Ext	Baixa	VS Code	Boa
QuickFix	Baixa	Muitas IDEs	Razoável
Snippets	Muito Baixa	Todas	Básica

Recomendação para Começar:

1. **Fase 1:** Implemente **snippets** (rápido e funciona agora)
2. **Fase 2:** Adicione **CLI** com `--inplace` para gerar no arquivo
3. **Fase 3:** Crie **VS Code Extension** (maior mercado)
4. **Fase 4:** Implemente **LSP Server** (cobre todas IDEs)

Implementação Rápida da Fase 2:

cpp

```
// cpp-serializer-gen --inplace Usuario.h
#include <fstream>
#include <sstream>

int main(int argc, char* argv[]) {
    if (argc > 1 && std::string(argv[1]) == "--inplace") {
        std::string filename = argv[2];

        // Lê arquivo
        std::ifstream file(filename);
        std::stringstream buffer;
        buffer << file.rdbuf();
        std::string content = buffer.str();

        // Encontra classe SERIALIZABLE
        size_t pos = content.find("SERIALIZABLE");
        if (pos != std::string::npos) {
            // Encontra fim da classe
            size_t class_end = content.find('}', pos);

            // Gera métodos
            std::string methods = generateMethods(content.substr(pos, class_end - pos));
        }
    }
}
```

```

    // Insere antes do '}'
    content.insert(class_end - 1, "\n" + methods);

    // Reescreve arquivo
    std::ofstream out(filename);
    out << content;

    std::cout << "✓ Generated serialization methods in " << filename << s
td::endl;
}
}

return 0;
}

```

Isso já dá uma experiência decente: usuário roda o comando e os métodos aparecem no arquivo!

Quer que eu detalhe alguma abordagem específica? A de VS Code Extension é bem popular e atinge muitos desenvolvedores.

```

/*
 * Qual a ideia:
 * 1 - Recebe o path do source do projeto por argv
 * 2 - Percorre as pastas em busca do .h ou .hpp
 * 3 - Se tiver o preprocessor SERIALIZABLE começa a fazer a mágica
 * 4 - Vai percorrer o arquivo mapeando tipos inteiros e objetos STL
 * 5 - Vai criar um arquivo e colocar seu include no cabeçalho desse cidadão
 * 6 - nesse arquivo vai implementar 'serialize' e 'deserialize'
 * *** No futuro quem sabe mapear pra criar reflection, seria fixe ***
 * 7 - Vai ignorar variáveis marcadas com 'TRANSIENT'
 * 8 - SOMENTE variáveis públicas serão serializadas
 */
ta bom isso

```

Perfeito! Essa especificação está excelente e bem pensada. Vou te ajudar a implementar passo a passo:



 Estrutura de Projeto

bash

```
cpp-serializer-gen/
├── CMakeLists.txt
└── src/
    ├── main.cpp          # Ponto de entrada (seu algoritmo)
    ├── FileWalker.cpp    # Passo 2: Percorrer pastas
    ├── Parser.cpp        # Passo 3-4: Analisar arquivos .h/.hpp
    ├── CodeGenerator.cpp # Passo 5-6: Gerar código
    └── TypeChecker.cpp   # Verificar tipos serializáveis
└── include/
    └── cpp-serializer-gen/
        ├── Macros.h      # SERIALIZABLE e TRANSIENT macros
        └── Serializer.h   # Interface de serialização
└── examples/           # Exemplos de uso
```

 Implementação Passo a Passo

Passo 1: main.cpp (Argument parsing)

cpp

```
// main.cpp
#include <iostream>
#include <filesystem>
#include <string>
#include "FileWalker.h"
#include "Parser.h"
#include "CodeGenerator.h"

namespace fs = std::filesystem;

int main(int argc, char* argv[]) {
    if (argc != 2) {
        std::cerr << "Uso: " << argv[0] << " <caminho-do-projeto>\n";
        return 1;
    }

    fs::path projectPath = argv[1];
    if (!fs::exists(projectPath)) {
        std::cerr << "Erro: Caminho não existe: " << projectPath << "\n";
        return 1;
    }
```

```
std::cout << "🔍 Analisando projeto: " << projectPath << "\n";

// Passo 2: Encontrar todos arquivos .h/.hpp
FileWalker walker;
auto headerFiles = walker.findHeaderFiles(projectPath);

std::cout << "📁 Encontrados " << headerFiles.size() << " arquivos header\n";

// Processar cada arquivo
for (const auto& headerFile : headerFiles) {
    std::cout << "📝 Processando: " << headerFile.filename() << "\n";

    // Passo 3: Verificar se tem SERIALIZABLE
    Parser parser;
    if (parser.containsSerializableMacro(headerFile)) {
        std::cout << "✨ Contém SERIALIZABLE!\n";

        // Passo 4: Extrair informações da classe
        auto classInfo = parser.parseClass(headerFile);

        if (classInfo) {
            std::cout << "📋 Classe: " << classInfo->name
                << " com " << classInfo->fields.size() << " campos\n";

            // Passo 5-6: Gerar arquivo de serialização
            CodeGenerator generator;
            auto generatedFile = generator.generateSerializationFile(
                *classInfo,
                projectPath
            );

            if (generatedFile) {
                std::cout << "✅ Gerado: " << *generatedFile << "\n";

                // Adicionar include no header original (opcional)
                generator.addIncludeToHeader(headerFile, *generatedFile);
            }
        }
    }
}

std::cout << "🎉 Concluído!\n";
return 0;
}
```

Passo 2: FileWalker.h/cpp

```
cpp

// FileWalker.h
#pragma once
#include <filesystem>
#include <vector>

class FileWalker {
public:
    std::vector<std::filesystem::path> findHeaderFiles(
        const std::filesystem::path& projectPath
    ) const;

private:
    bool isHeaderFile(const std::filesystem::path& path) const;
};

// FileWalker.cpp
#include "FileWalker.h"
#include <iostream>

namespace fs = std::filesystem;

std::vector<fs::path> FileWalker::findHeaderFiles(
    const fs::path& projectPath
) const {
    std::vector<fs::path> headers;

    try {
        for (const auto& entry : fs::recursive_directory_iterator(projectPath)) {
            if (entry.is_regular_file() && isHeaderFile(entry.path())) {
                headers.push_back(entry.path());
            }
        }
    } catch (const fs::filesystem_error& e) {
        std::cerr << "Erro ao percorrer diretórios: " << e.what() << "\n";
    }

    return headers;
}

bool FileWalker::isHeaderFile(const fs::path& path) const {
    std::string ext = path.extension().string();
    return ext == ".h" || ext == ".hpp" || ext == ".hxx" || ext == ".hh";
```

}

Passo 3-4: Parser.h/cpp

```
cpp

// Parser.h
#pragma once
#include <filesystem>
#include <optional>
#include <string>
#include <vector>

struct FieldInfo {
    std::string type;           // "int", "std::string", "std::vector<double>"
    std::string name;           // "id", "nome", "valores"
    bool isPublic;              // Passo 8: Somente públicas
    bool isTransient;           // Passo 7: Marcado com TRANSIENT
};

struct ClassInfo {
    std::string name;           // "Usuario", "Produto"
    std::vector<FieldInfo> fields;
    std::filesystem::path sourceFile;
};

class Parser {
public:
    // Passo 3: Verificar macro SERIALIZABLE
    bool containsSerializableMacro(const std::filesystem::path& filePath) const;

    // Passo 4: Extraír informações da classe
    std::optional<ClassInfo> parseClass(const std::filesystem::path& filePath) const;

private:
    std::string readFile(const std::filesystem::path& filePath) const;
    std::vector<std::string> splitLines(const std::string& content) const;
    bool isSerializableType(const std::string& type) const;
    AccessSpecifier currentAccessFromLine(const std::string& line) const;
};
```

Passo 5-6: CodeGenerator.h/cpp

cpp

```
// CodeGenerator.h
#pragma once
#include <filesystem>
#include "ClassInfo.h"

class CodeGenerator {
public:
    // Passo 5: Criar arquivo com include
    std::optional<std::filesystem::path> generateSerializationFile(
        const ClassInfo& classInfo,
        const std::filesystem::path& projectPath
    );

    // Adicionar include ao header original
    bool addIncludeToHeader(
        const std::filesystem::path& headerFile,
        const std::filesystem::path& generatedFile
    );
};

private:
    std::string generateHeaderContent(const ClassInfo& classInfo) const;
    std::string generateSerializeMethod(const ClassInfo& classInfo) const;
    std::string generateDeserializeMethod(const ClassInfo& classInfo) const;
    std::filesystem::path getOutputPath(
        const ClassInfo& classInfo,
        const std::filesystem::path& projectPath
    ) const;
};
```



Macros.h para os usuários

cpp

```
// Macros.h - Para os usuários incluirem
#pragma once

// Passo 3: Marcador para classes serializáveis
#define SERIALIZABLE(ClassName) /* Marcador para o parser */

// Passo 7: Para campos que NÃO devem ser serializados
#ifndef __clang__
    #define TRANSIENT [[maybe_unused]]
#endif
#ifndef __GNUC__
    #define TRANSIENT [[gnu_transient]]
#endif
```

```
#define TRANSIENT [[maybe_unused]]  
#elif defined(_MSC_VER)  
    #define TRANSIENT [[maybe_unused]]  
#else  
    #define TRANSIENT /* nothing */  
#endif
```

Exemplo de Uso

Arquivo do usuário:

cpp

```
// Usuario.h  
#pragma once  
#include <string>  
#include <vector>  
  
SERIALIZABLE(Usuario) // Passo 3: Marcador encontrado pelo parser  
class Usuario {  
public: // Passo 8: Somente públicos  
    int id; // ✓ Serializado  
    std::string nome; // ✓ Serializado  
    std::vector<std::string> emails; // ✓ Serializado  
  
    TRANSIENT // Passo 7: Ignorado  
    std::string sessionToken; // ✗ NÃO serializado  
  
private: // Passo 8: Privados ignorados  
    std::string senhaHash; // ✗ NÃO serializado  
  
public:  
    double salario; // ✓ Serializado  
};
```

Arquivo gerado pelo seu tool:

cpp

```
// Usuario_serialization.h (gerado automaticamente)  
#pragma once  
#include "Usuario.h"  
#include <nlohmann/json.hpp>
```

```
#include <vector>
#include <string>

// Serialização para JSON
template<>
struct JsonSerializer<Usuario> {
    static nlohmann::json to_json(const Usuario& obj) {
        return {
            {"id", obj.id},
            {"nome", obj.nome},
            {"emails", obj.emails},
            {"salario", obj.salario}
            // sessionToken omitido (TRANSIENT)
            // senhaHash omitido (privado)
        };
    }

    static Usuario from_json(const nlohmann::json& j) {
        Usuario obj;
        obj.id = j["id"];
        obj.nome = j["nome"];
        obj.emails = j["emails"].get<std::vector<std::string>>();
        obj.salario = j["salario"];
        // sessionToken e senhaHash mantêm valores default
        return obj;
    }
};

// Também pode gerar serialização binária
// template<>
// struct BinarySerializer<Usuario> { ... };
```

Header modificado com include:

cpp

```
// Usuario.h (após execução do tool)
#pragma once
#include <string>
#include <vector>
#include "Usuario_serialization.h" // Passo 5: Include adicionado

SERIALIZABLE(Usuario)
class Usuario {
public:
    ..
```

```
// ... campos originais ...

// Passo 6: Métodos serialização disponíveis via include
// JsonSerializer<Usuario>::to_json(usuario);
// JsonSerializer<Usuario>::from_json(json);
};
```

🔍 Algoritmo de Parsing Simplificado

cpp

```
// Parser.cpp - parseClass simplificado
std::optional<ClassInfo> Parser::parseClass(const fs::path& filePath) const {
    std::string content = readFile(filePath);
    std::vector<std::string> lines = splitLines(content);

    ClassInfo classInfo;
    classInfo.sourceFile = filePath;

    bool inClass = false;
    bool inPublicSection = false;

    for (size_t i = 0; i < lines.size(); i++) {
        std::string line = trim(lines[i]);

        // Encontrar SERIALIZABLE(ClassName)
        if (line.find("SERIALIZABLE") != std::string::npos) {
            size_t start = line.find('(') + 1;
            size_t end = line.find(')');
            classInfo.name = line.substr(start, end - start);
            inClass = true;
            continue;
        }

        if (!inClass) continue;

        // Verificar seção pública/privada
        if (line == "public:") {
            inPublicSection = true;
        } else if (line == "private:" || line == "protected:") {
            inPublicSection = false;
        }

        // Encontrar fim da classe
        if (line == "};" || line == "};") {
            break;
        }
    }
}
```

```

    ui con,
}

// Parse de campos (simplificado)
if (inPublicSection && !line.empty() && line.back() == ';') {
    auto field = parseField(line);
    if (field) {
        classInfo.fields.push_back(*field);
    }
}
}

return classInfo;
}

```

Build com CMake

cmake

```

cmake_minimum_required(VERSION 3.15)
project(cpp-serializer-gen VERSION 1.0.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Executável principal
add_executable(cpp-serializer-gen
    src/main.cpp
    src/FileWalker.cpp
    src/Parser.cpp
    src/CodeGenerator.cpp
    src/TypeChecker.cpp
)

```

Instalação

```

install(TARGETS cpp-serializer-gen DESTINATION bin)
install(FILES include/cpp-serializer-gen/Macros.h
    DESTINATION include/cpp-serializer-gen)

```

Plano de Implementação

1. Semana 1: FileWalker + argument parsing
2. Semana 2: Parser básico (encontra SERIALIZABLE)

3. **Semana 3:** TypeChecker (detecta tipos serializáveis)
4. **Semana 4:** CodeGenerator (gera arquivos .h)
5. **Semana 5:** Testes + tratamento de edge cases
6. **Semana 6:** Integração CMake + documentação

Checklist das Funcionalidades

- Passo 1: Receber path via argv ✓
- Passo 2: Percorrer pastas recursivamente ✓
- Passo 3: Detectar SERIALIZABLE macro ✓
- Passo 4: Mapear tipos inteiros e STL ✓
- Passo 5: Criar arquivo e adicionar include ✓
- Passo 6: Implementar serialize/deserialize ✓
- Passo 7: Ignorar variáveis TRANSIENT ✓
- Passo 8: Somente variáveis públicas ✓

Sua especificação está ótima! É simples, direta e resolve um problema real. O único detalhe que você pode querer adicionar depois é **suporte a namespaces** (classes dentro de `namespace MeuNamespace { ... }`).

Quer que eu detalhe alguma parte específica, como o algoritmo de parsing ou a geração do código?