EXERCÍCIO - PROGRAMAÇÃO ORIENTADA A OBJETOS

PROF. CARLOS EDUARDO BATISTA

HERANÇA, SOBRECARGA DE OPERADORES E VARIÁVEIS E FUNÇÕES DE CLASSE

Manual de Boas Práticas de Programação para o Sistema de Gerenciamento de Campeonato de Futebol

Objetivo do Manual

Este manual tem como objetivo orientar os alunos no uso de boas práticas de codificação em C++ para o desenvolvimento do exercício proposto. As orientações apresentadas visam garantir que o código produzido seja de fácil manutenção, leitura e que atenda aos padrões de desenvolvimento esperados para um ambiente profissional.

1. Estilo de Codificação

- **Nomes de Variáveis e Funções:**
  - Utilize `snake_case` para nomear variáveis, funções e métodos.
  - Nomes devem ser claros e descritivos, indicando claramente o propósito da variável ou função.
  - Exemplo:

```
int numero_gols;            // Correto: nome claro e em snake_case
void calcular_pontuacao();  // Correto: nome de função descritivo e em
snake_case
```

- **Nomes de Classes:**
  - Utilize `snake_case` para nomes de classes (inicial minúscula e palavras separadas por _).
  - Os nomes devem indicar claramente o papel ou responsabilidade da classe.
  - Exemplo:

```
class tecnico_jogador;      // Correto: nome de classe em snake_case
class campeonato_futebol;   // Correto: nome de classe em snake_case
```

- **Nomes de Constantes:**
  - Utilize `UPPER_CASE` para constantes.
  - Exemplo:

```
const int MAX_JOGADORES = 11;
const double BONUS_TECNICO_JOGADOR = 500.0;
```

2. Organização de Arquivos

- **Separação entre Cabeçalho (`.h`) e Implementação (`.cpp`):**
  - Defina as classes e suas interfaces nos arquivos `.h` (cabeçalhos).
  - Implemente as funções e métodos nos arquivos `.cpp` correspondentes.
  - Essa separação facilita a manutenção e a compilação do código.
- **Nomenclatura dos Arquivos:**
  - Nomeie os arquivos de cabeçalho e código-fonte utilizando o nome da classe correspondente.
  - Exemplo:
    - `tecnico_jogador.h` e `tecnico_jogador.cpp`
    - `campeonato_futebol.h` e `campeonato_futebol.cpp`

3. Documentação do Código

- **Comentários Padrão C++:**
  - Utilize `//` para comentários curtos e explicativos dentro de funções.
  - Use `/* ... */` para comentários de bloco e para explicar a funcionalidade de grupos de código.
- **Documentação de Classes e Métodos:**
  - Documente cada classe e método utilizando o padrão de comentários do Doxygen (ou similar) para facilitar a geração automática de documentação.
  - Inclua descrições de parâmetros e valores de retorno para cada método.
  - Exemplo:

```
/**
 * @brief Classe que representa um jogador de futebol.
 *
 * A classe jogador mantém informações sobre a posição, gols marcados
 * e salário do jogador, além de permitir o cálculo do salário total.
 */
class jogador {
    private:
        std::string posicao;
        int gols_marcados;

    public:
        /**
         * @brief Construtor padrão que inicializa um jogador com uma
posição e número de gols.
         *
         * @param posicao A posição do jogador (goleiro, zagueiro,
etc.).
         * @param gols Número de gols marcados pelo jogador.
         */
        jogador(const std::string &posicao, int gols);
```

```
        /**
         * @brief Calcula o salário total do jogador com base no número
de gols.
         *
         * @return double Salário total calculado.
         */
        double calcular_salario();
};
```

4. Organização das Classes e Métodos

- **Defina uma Classe por Arquivo**:
  - Cada classe deve ser declarada e implementada em arquivos separados.
  - Use arquivos `.h` para definir a interface da classe e `.cpp` para implementar os métodos.
- **Ordem de Declaração**:
  - Na declaração da classe (arquivo `.h`), siga a ordem:
    - Atributos privados e protegidos.
    - Construtores e destrutores.
    - Métodos públicos.
    - Métodos privados e protegidos.
  - Use modificadores de acesso (`private`, `protected` e `public`) para delimitar os grupos de atributos e métodos.

5. Tratamento de Erros e Controle de Fluxo

- **Verificações de Consistência:**
  - Utilize `assert()` para validar pré-condições e pós-condições dos métodos.
  - Se necessário, use condicionais (`if`) para verificar valores inválidos e tratar erros de forma apropriada.
  - Exemplo:

```
void adicionar_gols(int gols) {
    assert(gols >= 0); // Gols não podem ser negativos.
    this->gols_marcados += gols;
}
```

6. Boas Práticas para Métodos

- **Métodos Constantes:**
  - Sempre que possível, marque métodos que não alteram o estado interno de uma classe como `const`.
  - Isso permite que métodos de leitura sejam utilizados em contextos onde o estado do objeto não deve ser modificado.

- Exemplo:

```
int get_gols_marcados() const;
```

- **Passagem de Parâmetros por Referência Constante:**
  - Use referência constante (`const &`) para passagem de objetos como parâmetros, evitando cópias desnecessárias.
  - Exemplo:

```
void set_tecnico(const tecnico &t);
```

7. Formatação e Indentação

- **Indentação Consistente:**
  - Utilize 4 espaços por nível de indentação. Não use tabulações.
  - Exemplo:

```
void calcular_pontuacao() {
    if (vitorias > 0) {
        pontuacao += vitorias * 3;
    }
}
```

- **Espaçamento e Quebra de Linhas:**
  - Utilize uma linha em branco entre funções e blocos de código lógico para melhorar a legibilidade.
  - Deixe um espaço entre os operadores e os operandos:

```
pontuacao = gols_time_casa + gols_time_visitante;
```

8. Outras Boas Práticas

- **Evite Uso de Literais Mágicos**:
  - Use constantes para representar valores fixos no código.
  - Exemplo:

```
const int MAX_PARTIDAS = 38; // Número máximo de partidas em um
campeonato.
```

- **Use Inicializadores de Lista**:
    - Para inicializar membros da classe, prefira inicializadores de lista no construtor:

```
jogador::jogador(const std::string &posicao, int gols)
    : posicao(posicao), gols_marcados(gols) {}
```

Organização dos Diretórios

Aqui está a organização sugerida dos diretórios e arquivos do projeto:

```
campeonato_futebol/
├── include/                    # Diretório para arquivos de cabeçalho (.h)
│   ├── pessoa.h
│   ├── membro_clube.h
│   ├── jogador.h
│   ├── tecnico.h
│   ├── tecnico_jogador.h
│   ├── juiz.h
│   ├── time.h
│   ├── jogo.h
│   └── campeonato.h
├── src/                        # Diretório para arquivos de implementação (.cpp)
│   ├── pessoa.cpp
│   ├── membro_clube.cpp
│   ├── jogador.cpp
│   ├── tecnico.cpp
│   ├── tecnico_jogador.cpp
│   ├── juiz.cpp
│   ├── time.cpp
│   ├── jogo.cpp
│   ├── campeonato.cpp
│   └── main.cpp                # Arquivo principal que executa o sistema
├── Makefile                     # Arquivo Makefile para compilar o projeto
└── README.md                    # Arquivo README com instruções do projeto
```

- **`include/`**: Contém todos os arquivos de cabeçalho (`.h`) para definir as classes e interfaces do projeto.
- **`src/`**: Contém todos os arquivos de implementação (`.cpp`), incluindo o `main.cpp` para o ponto de entrada do programa.
- **`Makefile`**: Arquivo `Makefile` para gerenciar a compilação do projeto.

Conteúdo do Makefile

Aqui está o `Makefile` completo para compilar o projeto:

```
# Makefile para o Sistema de Gerenciamento de Campeonato de Futebol

# Diretórios
INCLUDE_DIR = include
SRC_DIR = src
BUILD_DIR = build

# Compilador e flags
CXX = g++
CXXFLAGS = -Wall -Wextra -std=c++17 -I$(INCLUDE_DIR)

# Arquivos de cabeçalho e código-fonte
HEADERS = $(wildcard $(INCLUDE_DIR)/*.h)
SOURCES = $(wildcard $(SRC_DIR)/*.cpp)

# Objetos gerados para cada arquivo-fonte
OBJECTS = $(patsubst $(SRC_DIR)/%.cpp, $(BUILD_DIR)/%.o, $(SOURCES))

# Nome do executável
EXEC = campeonato_futebol

# Alvo padrão para compilar todo o projeto
all: $(BUILD_DIR) $(EXEC)

# Cria o diretório build, caso não exista
$(BUILD_DIR):
    mkdir -p $(BUILD_DIR)

# Gera o executável a partir dos objetos
$(EXEC): $(OBJECTS)
    $(CXX) $(CXXFLAGS) -o $@ $^

# Regra para compilar os arquivos-fonte em arquivos-objeto
$(BUILD_DIR)/%.o: $(SRC_DIR)/%.cpp $(INCLUDE_DIR)/%.h
    $(CXX) $(CXXFLAGS) -c $< -o $@

# Limpeza dos arquivos compilados
clean:
    rm -rf $(BUILD_DIR) $(EXEC)

# Executa o programa após compilar
run: all
    ./$(EXEC)

# Recompila o projeto
rebuild: clean all
```

Explicação do `Makefile`:

1. **Diretórios Definidos:**
    - `INCLUDE_DIR = include`: Diretório para os arquivos de cabeçalho (`.h`).

- `SRC_DIR = src`: Diretório para os arquivos de código-fonte (`.cpp`).
- `BUILD_DIR = build`: Diretório para armazenar os arquivos objeto (`.o`) gerados durante a compilação.

2. **Compilador e Flags:**

- `CXX = g++`: Define o compilador C++ (GCC).
- `CXXFLAGS = -Wall -Wextra -std=c++17 -I$(INCLUDE_DIR)`: Flags para o compilador, incluindo:
  - `-Wall` e `-Wextra` para habilitar todos os warnings.
  - `-std=c++17` para definir o padrão C++ 17.
  - `-I$(INCLUDE_DIR)` para incluir o diretório de cabeçalhos (`include`).

3. **Arquivos e Objetos:**

- `HEADERS = $(wildcard $(INCLUDE_DIR)/*.h)`: Busca todos os arquivos `.h` no diretório `include`.
- `SOURCES = $(wildcard $(SRC_DIR)/*.cpp)`: Busca todos os arquivos `.cpp` no diretório `src`.
- `OBJECTS = $(patsubst $(SRC_DIR)/%.cpp, $(BUILD_DIR)/%.o, $(SOURCES))`: Converte cada arquivo `.cpp` em um arquivo `.o` no diretório `build`.

4. **Alvo Padrão (`all`):**

- Define a construção completa do projeto, garantindo que o diretório `build` exista e compilando todos os objetos.

5. **Regra para Compilar Arquivos:**

- `$(BUILD_DIR)/%.o: $(SRC_DIR)/%.cpp $(INCLUDE_DIR)/%.h`: Compila cada arquivo `.cpp` do diretório `src` em um arquivo `.o` no diretório `build`.

6. **Alvo `clean`:**

- Remove os arquivos gerados (`.o` e executável) e limpa o diretório `build`.

7. **Alvo `run`:**

- Compila e executa o programa após a compilação.

8. **Alvo `rebuild`:**

- Remove os arquivos gerados e recompila o projeto do zero.

Como Utilizar o `Makefile`

1. **Compilar o Projeto:**

- No terminal, navegue até a pasta do projeto e execute:

```
make
```

2. **Executar o Programa:**
    - Após compilar, execute o programa com:

```
make run
```

3. **Limpar Arquivos Gerados:**
    - Para remover todos os arquivos gerados e o executável:

```
make clean
```

4. **Recompilar o Projeto:**
    - Para limpar e recompilar todo o projeto:

```
make rebuild
```