UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO E ENGENHARIA DE COMPUTAÇÃO

LEONARDO DALCIN
PAULO RICARDO RAMOS DA ROSA
PIETRO DEGRAZIA

Jogo Pedagógico em C++

Relatório apresentado como requisito parcial para a obtenção de conceito na Disciplina de Modelos de Linguagens de Programação

Prof. Dr. Lucas Mello Schnorr
Orientador

Porto Alegre
2018

SUMÁRIO

1 INTRODUÇÃO

Este trabalho tem como objetivo a implementação em C++ de um jogo pedagógico que visa ensinar diversos conceitos da cadeira de Modelos de Linguagens de Programação em diferentes níveis de profundidade. O programa permitirá que o usuário compartilhe seus resultados e confira respostas de outros colegas, assim como explicações e links para referências sobre o assunto.

1.1 Projeto

A proposta do projeto consta em, dada uma linguagem de programação escolhida pelo grupo dentre as pré-selecionadas pelo professor, desenvolver uma aplicação com duas implementações: uma puramente orientada a objetos e outra puramente funcional, contendo alguma forma de paralelismo. C++17 será a linguagem utilizada para ambas as versões.

1.2 Sobre o jogo

O projeto será um jogo de perguntas e respostas sobre a disciplina de Modelos de Linguagem de Programação, aos moldes dos programas "Show do Milhão"e "Quem quer ser um milionário?", com a objetivo de ajudar o estudante na disciplina. O jogo contará com explicações sobre o conteúdo das perguntas e poderá ser feito a consulta das respostas dos outros jogadores.

2 C++17

2.1 Origem

A linguagem C++ foi introduzida inicialmente em 1985 por Bjarne Stroustrup, herdando as características da linguagem C mas adicionando o conceito de Orientação a Objetos. Desde então, passou por diversas atualizações, estando hoje padronizada na versão ISO/IEC 14882:2017, informalmente conhecida como C++17.

2.2 Programa

Programas em C++ podem conter valores, objetos, referências, funções, enumeradores, tipos, membros de classes, templates, especializações de template e namespaces. Essas entidades são criadas através de declarações, que associam a entidade com um nome e define suas propriedades. A declaração que define todas as propriedades para usar uma entidade é chamada de definição.

Nomes encontrados em um programa são associados com a declaração em que aparecem. Eles somente são válidos no escopo em que foram declarados. Alguns nomes, dependendo da linkagem, podem referênciar entidades que aparecem em um escopo diferente.

2.3 Objetos

Programas em C++ criam, destroem, referenciam, acessam e manipulam objetos. Um objeto em C++ é uma região de armazenamento que possui:

- tamanho(sizeof);
- requisito de alinhamento(alignof);
- duração de alocação(automática, estática, dinâmica, local);
- duração de vida(baseada na alocação ou temporária);
- tipo;
- valor(pode ser indeterminado);

- nome(opcional).

Objetos são criados com definições, expressões new, expressões throw, por manipulação de membros de uma union e onde objetos temporários são necessários.

2.4 Classes

Classes e estruturas são tipos definidos pelo usuário, usando especificadores de classe, que devem aparecer na sequencia a seguir.

- Chave de classe - pode ser class ou struct, a única diferença entre elas é o acesso padrão, para estruturas é acesso a membro, e para classes é a membro da classe base.
- Atributos - sequencia opcional de atributos, noreturn, por exemplo. Também pode ser incluido especificadores de alinhamento.
- Nome - o identificador da classe. Opcionalmente acompanhado da palavra chave final. Caso o nome seja omitido, a classe é considerada anônima e não pode ser final.
- Classes base - lista opcional de classes pai e o modelo de herança que será usado para cada uma delas.
- Membros - lista de membros, especificadores de acesso, declaração e definição de funções de instância, tipos e classes aninhados.

2.5 Classes Abstratas

Classes abstratas são tipos que não podem ser instanciados, mas podem ser usados como classe base. Para criar uma, basta usar o especificador virtual. Classes como essas são usadas para representar conceitos gerais, por exemplo, Forma ou Animal, esses tipos serão usados como classe base para tipos concretos como Círculo ou Cachorro.

Não é possível usar tipos abstratos em parâmetros, tipos de retorno ou conversões. É possível usar esses tipos com ponteiros e referências.

Classes abstratas em C++ são implementadas declarando pelo menos um de seus métodos como virtual puro. Um método virtual puro é especificado colocando "=0"na sua

declaração. O propósito de uma classe abstrata é prover uma base na qual outras classes podem herdar atributos e métodos. Se uma subclasse X herda de uma classe abstrata Y, ela precisa implementar todos os métodos virtuais puros, caso contrário ela será considerada uma classe abstrata também. Se ela foi instanciada, um erro de compilação ocorrerá.

2.6 Polimorfismo

Objetos de uma classe que declaram ou herdam pelo menos uma função virtual são considerados polimórficos. Dentro de cada objeto deste tipo, a implementação contém informações adicionais que são usadas por chamadas de funções virtuais e por métodos auxiliares como dynamic cast ou typeid.

2.6.1 Polimorfismo por Inclusão

Também chamado de polimorfismo de subtipo ou polimorfismo em tempo de execução, acontece durante a execução através de uma tabela virtual. O compilador não localiza o endereço da função a ser chamada em tempo de compilação, em vez disso quando o programa é executado, a função é referenciada por um ponteiro na tabela virtual.

2.6.2 Polimorfismo Paramétrico

Polimorfismo paramétrico possibilita executar um mesmo código para qualquer tipo. Em C++ polimorfismo paramétrico é implementado via templates. Um exemplo de template utilizado no projeto é o método **max** :

Listing 2.1 – Método genérico para calcular o máximo entre dois argumentos genéricos

```
1
2 #include <iostream>
3 #include <string>
4
5 template <class T>
6 T max(T a, T b) {
7     return a > b ? a : b;
```

```
8  }
9
10 int main() {
11     std::cout << ::max(9, 5) << std::endl;       // 9
12
13     std::string foo("foo"), bar("bar");
14     std::cout << ::max(foo, bar) << std::endl; // "foo"
15 }
```

Contudo, esse método não funciona em ponteiros porque comparar ponteiros é comparar o local de memória e não o conteúdo. Para funcionar com ponteiros, deve-se especificar um template para ponteiros, o que deixaria de ser polimorfismo paramétrico e passaria a ser ad-hoc. Polimorfismo paramétrico também é chamado de polimorfismo em tempo de compilação, já que ocorre durante a compilação.

2.7 Memória

C++ possui 4 tipos de gerenciamento de memória:

- Armazenamento de duração estática: são criados antes da chamada main() (salvo exceções) e destruídos na ordem reversa de criação após a saída de main().
- Armazenamento de duração de thread: Similar ao armazenamento estático, porém o objeto é criado com a thread e destruído com o join da thread.
- Armazenamento automático: Variáveis automáticas são criadas no ponto de declaração e destruídas na ordem reversa de criação de seu escopo. São alocadas automaticamente na pilha.
- Armazenamento dinâmico: São criadas com uma chamada new e destruídas com uma chamada delete.

3 IMPLEMENTAÇÃO

3.1 Orientação a Objetos

A implementação inicial inclui duas classes bases:

- Player: Classe que será utilizada para criar informações sobre o jogador, como nome e pontuação.

  Atributos: std::string name, int score.

  Métodos: Construtor, getScore, setScore(int score).

- Question: Classe que conterá uma pergunta com suas alternativas e resposta.

  Atributos: std::string enunciation, std::string alternatives[4], int answer.

  Métodos: Construtor, showQuestion(): mostra a pergunta e as alternativas, respond(int alternative): recebe a resposta do usuário, askQuestion(): realiza a pergunta com os métodos showQuestion() e respond(int alternative).

Além dessas classes, uma classe Database está incluída para gravar dados do programa, essencialmente as perguntas, e uma classe Main para execução do programa.

3.2 Recursos utilizados para a implementação OO

- Recursos mínimos sugeridos

  1. Especificar e utilizar classes (utilitárias ou para representar as estruturas de dados utilizadas pelo programa).

Listing 3.1 – Exemplo de classe utilizada no programa

```
1  #include <iostream>
2  #include <string>
3
4  class Player
5  {
6  private:
7  std::string name;
8  int score;
```

```
9       public:
10      Player(std::string name){
11      this->name = name;
12      }
13      int getScore()
14      {
15      return this->score;
16      }
17      void setScore(int score)
18      {
19      this->score = score;
20      }
21      };
```

2. Fazer uso de encapsulamento e proteção dos atributos, com os devidos métodos de manipulação (setters/getters) ou propriedades de acesso, em especial com validação dos valores (parâmetros) para que estejam dentro do esperado ou gerem exceções caso contrário.

Observando o Listing 3.1 o atributo **score** é protegido pelos métodos **getScore** e **setScore**, impossibilitando algum desenvolvedor de ter acesso direto à esta propriedade do objeto **Player**. Isso centraliza a leitura e a escrita desse atributo, resultando em uma melhor organização do código.

Além disso, seções do programa que precisam de entrada de dados do usuário estão devidamente protegidas contra dados indevidos, especificamente na entrada da resposta quando um laço *while* impede dados inválidos. Caso a entrada não seja **int**, ela é limpa e ignorada.

Listing 3.2 – Proteção de dados inválidos

```
1       while ( !(std::cin >> answer)) {
2       std::cin.clear();
3       std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\
4       std::cout << "Valor invalido\n";
5       }
```

3. Especificação e uso de construtores-padrão para a inicialização dos atributos e, sempre que possível, de construtores alternativos.

O listing 3.1 também demonstra o uso de um construtor não padrão que necessita somente do atributo **name** que tem especificação de tipo, garantindo então a tipagem correta deste parametro.

4. Especificação e uso de destrutores (ou métodos de finalização), quando necessário. Destrutores foram especificados para mostrar uma mensagem caso sejam chamados.

Listing 3.3 – Exemplo de destrutor

```
1    Player::~Player()
2    {
3    std::cout << "Jogador deletado" << std::endl;
4    }
```

5. Organizar o código em espaços de nome diferenciados, conforme a função ou estrutura de cada classe ou módulo de programa.

As classes estão separadas em arquivos .h e .cpp, para cada classe ter seu determinado espaço de nome.

Listing 3.4 – Separação entre definição e implementação

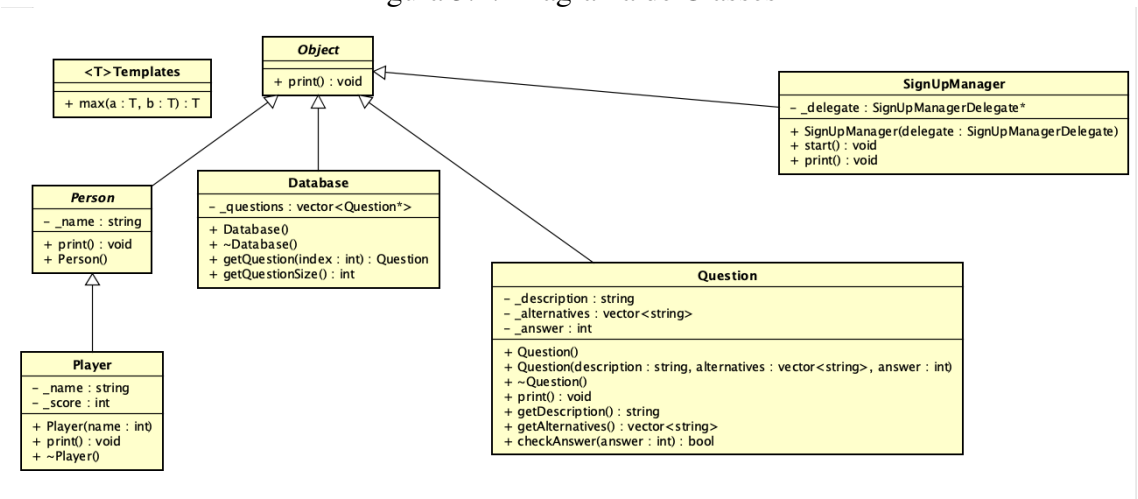
```
1    //Player.h
2    #ifndef MLPQUIZAPP_PLAYER_H
3    #define MLPQUIZAPP_PLAYER_H
4
5    #include "Person.h"
6    #include <iostream>
7    #include <string>
8    class Player:public Person
9    {
10   private:
11   int _score;
12   public:
13   Player(std::string name);
14   ~Player();
```

```
15      int getScore()const;
16      void setScore(int score);
17      void print();
18      };
19
20      #endif //MLPQUIZAPP_PLAYER_H
21
22      //Player.cpp
23      #include "Player.h"
24      #include <iostream>
25      #include <string>
26
27
28      Player::Player(std::string name):Person(name) {
29      this->_score = 0;
30      }
31
32      Player::~Player()
33      {
34      std::cout << "Jogador deletado" << std::endl;
35      }
36
37      void Player::setScore(int score) {
38      if (score < 0) { return; }
39      this->_score = score;
40      }
41
42      void Player::print() {
43      std::cout<< "Meu nome e "<< this->getName() << " e minha pontua
44      }
45      int Player::getScore() const
46      {
47      return _score;
48      }
```

6. Usar mecanismo de herança, em especial com a especificação de pelo menos três níveis de hierarquia, sendo pelo menos um deles correspondente a uma classe abstrata, mais genérica, a ser implementada nas classes-filhas.

O projeto possui uma classe topo chamada **Object**, da qual todas as outras classes são derivadas. Ela possui um método **print** a ser sobrescrito pelas subclasses, funcionando de maneira similar ao *toString* do Java. A classe abstrata **Person** é subclasse de Object e possui um atributo **name**, bem como uma implementação própria de **print**. A classe **Player** é subclasse de **Person**, herdando o atributo **name** e além de sobrescrever o método **print**, possui o atributo **score**, referente a pontuação do jogo.

Figura 3.1: Diagrama de Classes



7. Utilizar polimorfismo por inclusão (variável ou coleção genérica manipulando entidades de classes filhas, chamando métodos ou funções específicas correspondentes)

O método **print** presente em todas as classes que herdam de **Object** é um exemplo de polimorfismo por inclusão, já que a classe **Person** possui uma implementação e **Player** possui outra.

Listing 3.5 – Exemplo de polimorfismo por inclusão

```
1    //Classe Person
2    #include "Person.h"
3
4    std::string Person::getName()
5    {
```

```
6       return _name;
7       }
8
9       Person::Person(std::string name)
10      {
11      this->_name = name;
12
13      }
14
15      void Person::print()
16      {
17      std::cout<< "Meu nome e "<< this->_name << std::endl;
18      }
19
20      //Classe Player
21      #include "Player.h"
22      #include <iostream>
23      #include <string>
24
25
26      Player::Player(std::string name):Person(name) {
27      this->_score = 0;
28      }
29
30      Player::~Player()
31      {
32      std::cout << "Jogador deletado" << std::endl;
33      }
34
35      void Player::setScore(int score) {
36      if (score < 0) { return; }
37      this->_score = score;
38      }
39
```

```
40      void Player::print() {
41      std::cout<< "Meu nome e "<< this->getName() <<
42       " e minha pontuacao e " << this->getScore() << std::endl;
43      }
44      int Player::getScore() const
45      {
46      return _score;
47      }
```

8. Usar polimorfismo paramétrico através da especificação de *algoritmo* (método ou função genérico) utilizando o recurso oferecido pela linguagem (i.e., generics, templates ou similar) e da especificação de *estrutura de dados* genérica utilizando o recurso oferecido pela linguagem.

A classe **Templates** é uma classe genérica com o método **max** para calcular o máximo entre dois valores genéricos. Ela é usada para identificar se o valor inserido para a alternativa não excede o limite de alternativas, o que causaria um erro.

Listing 3.6 – Proteção de dados inválidos

9. Usar polimorfismo por sobrecarga (vale construtores alternativos).

A classe **Question** possui dois construtores: Um padrão sem argumentos, para construir uma pergunta genérica e outro com os parâmetros de enunciado, alternativas e resposta.

Listing 3.7 – Construtores da classe Question

```
1       Question::Question() {
2       this->_description = "Description";
3       this->_answer = 0;
4
5       std::vector<std::string> alternatives {"Option 1", "Option 2"};
6       this->_alternatives = alternatives;
7       };
8
9       Question::Question(std::string description, std::vector<std::st
10      this->_description = description;
```

```
11      this->_alternatives = alternatives;
12      this->_answer = answer;
13      }
```

10. Especificar e usar delegates.

Delegates são utilizados na classe **SignUpManager** para cadastrar um jogador com o **SignUpManagerDelegate**:

Listing 3.8 – Uso de delegate

```
1       typedef void SignUpManagerDelegate(Player);
2
3       class SignUpManager:Object {
4       public:
5       SignUpManager(SignUpManagerDelegate delegate);
6       void start();
7       void print();
8       private:
9       SignUpManagerDelegate* _delegate;
10      };
11
12      SignUpManager::SignUpManager(SignUpManagerDelegate delegate) {
13      this->_delegate = delegate;
14      };
15
16      void SignUpManager::start() {
17      std::string name;
18      std::cout << "Escolha um nome de usuario: ";
19      std::getline(std::cin, name);
20
21      auto player = Player(name);
22      this->_delegate(player);
23      }
```

- Recursos de processamento paralelo

1. Definição, uso e gerência de unidades (threads, módulos, classes, métodos, funções, trechos ou instruções) de execução concorrente e o seu sincronismo TODO

2. Definição, uso e gerência de regiões críticas (variáveis, arrays, coleções ou similares) TODO

3.3 Funcional

3.4 Recursos utilizados para a implementação Funcional

- Recursos mínimos sugeridos

  1. Priorizar o uso de elementos imutáveis e funções puras (por exemplo, sempre precisar manipular listas, criar uma nova e não modificar a original, seja por recursão ou através de funções de ordem maior). TODO

  2. Especificar e usar funções não nomeadas (ou lambda) TODO.

  3. Especificar e usar funções que usem currying TODO.

  4. Especificar funções que utilizem pattern matching ao máximo, na sua definição TODO.

  5. Especificar e usar funções de ordem superior (maior) criadas pelo programador TODO.

  6. Usar funções de ordem maior prontas (p.ex., map, reduce, foldr/foldl ou similares) TODO.

  7. Especificar e usar funções como elementos de 1ª ordem TODO.

  8.Usar recursão como mecanismo de iteração (pelo menos em funções de ordem superior que manipulem listas) TODO.

- Recursos extras

  Exemplo de recurso extra

- Recursos de processamento paralelo

  1. Definição, uso e gerência de unidades (threads, módulos, classes, métodos, funções, trechos ou instruções) de execução concorrente e o seu sincronismo TODO

  2. Definição, uso e gerência de regiões críticas (variáveis, arrays, coleções ou similares) TODO

3.5 Análise da Linguagem

Uma análise da linguagem foi feita considerando diversos aspectos, como usabilidade, simplicidade, tipos de dados, entre outros.

3.6 Conclusões

A linguegem C++ é muito boa para ser utilizada no contexto de orientação a objetos ou programação procedural. Como é uma linguagem derivada do C, possui muitas ferramentas herdadas e facilitou o uso de muitas operações com a adição de bibliotecas (std::vector e std::string, por exemplo). Porém, em programação funcional a tarefa se tornou bem difícil, pois a sintaxe é bastante complicada de entender e as bibliotecas de suporte não são otimizadas para esse paradigma, causando muita dificuldade para implementar funções de simples comportamento. Contudo, após as funções iniciais serem criadas, o reuso é bem amplo. No geral, C++ é otimizada para aplicações utilizando orientação a objetos, com diversas ferramentas compatíveis(interfaces gráficas, motores de jogos,etc.), possuindo também suporte a programação funcional, mas não sendo a escolha ideal para tal, havendo linguagens mais bem estruturadas para esse paradigma (como R e Haskell, por exemplo).

3.7 Referências

- <https://en.cppreference.com/w/cpp/language/object>: Objects
- <https://en.cppreference.com/w/cpp/language/class>: Classes
- <https://en.cppreference.com/w/cpp/language/objectPolymorphic_objects>: Polymorphic Objects
- <https://en.cppreference.com/w/cpp/language/abstract_class>: Abstract Classes
- <http://www.cplusplus.com/info/history/>
- <http://www.catonmat.net/blog/cpp-polymorphism/>