

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

ANDRÉ DEXHEIMER  
GABRIEL PISCOYA  
RODRIGO WIEBBELLING

**Uma Comparação entre o Paradigma  
Funcional e Orientado a Objetos para  
Solução do Problema "Tower Defence"  
utilizando C++**

**Grupo HP**

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  
2017

## SUMÁRIO

<b>1 INTRODUÇÃO .....</b>	<b>3</b>
<b>1.1 Historia das Linguagens de Programação .....</b>	<b>3</b>
<b>1.2 Ambiente e Linguagem de Programação.....</b>	<b>4</b>
<b>1.3 Problema Abordado.....</b>	<b>4</b>
<b>2 A LINGUAGEM C++ .....</b>	<b>5</b>
<b>2.1 Algumas características do C++ .....</b>	<b>5</b>
<b>3 TOWER DEFENCE .....</b>	<b>7</b>
<b>3.1 Objetivo do jogo .....</b>	<b>7</b>
<b>3.2 Os inimigos .....</b>	<b>7</b>
<b>4 IMPLEMENTAÇÃO ORIENTADA A OBJETOS .....</b>	<b>8</b>
<b>4.1 Classes .....</b>	<b>8</b>
4.1.1 Classes Abstratas .....	8
4.1.2 Herança .....	9
4.1.3 Herança Unica.....	9
4.1.4 Herança Multipla .....	9
<b>4.2 Polimorfismo.....</b>	<b>10</b>
4.2.1 Polimorfismo Adhoc .....	10
4.2.2 Polimorfismo Universal .....	11
<b>4.3 Encapsulamento .....</b>	<b>12</b>
<b>4.4 Delegates .....</b>	<b>12</b>
<b>4.5 Espaço de Nomes.....</b>	<b>13</b>
<b>4.6 Construtores e Destrutores .....</b>	<b>14</b>
<b>4.7 Análise Crítica.....</b>	<b>15</b>
<b>5 IMPLEMENTAÇÃO FUNCIONAL .....</b>	<b>18</b>
<b>5.1 Funções Puras.....</b>	<b>18</b>
<b>5.2 Funções Lambda .....</b>	<b>18</b>
<b>5.3 Funções como elementos de 1ª ordem.....</b>	<b>19</b>
<b>5.4 Funções de ordem superior .....</b>	<b>20</b>
<b>5.5 Currying.....</b>	<b>20</b>
<b>5.6 Recursao.....</b>	<b>20</b>
<b>5.7 Pattern Matching .....</b>	<b>21</b>
<b>6 CONCLUSÃO .....</b>	<b>24</b>
<b>REFERÊNCIAS.....</b>	<b>25</b>

## 1 INTRODUÇÃO

Este capítulo tem o objetivo de descrever de forma sucinta a história das linguagens de programação e os principais tópicos envolvidos na realização deste trabalho. Logo após, serão abordados os temas diretamente relacionados à implementação, para maiores detalhes a implementação OO e funcional estão disponíveis em <https://github.com/andredxc/HPTowerDefense> e podem ser utilizadas sempre e quando a respectiva citação seja incluída. Caso seja necessário podemos fornecer uma máquina virtual do sistema de desenvolvimento.

### Historia das Linguagens de Programação

As primeiras linguagens de programação eram simples códigos utilizados para automatizar processos nem sempre relacionadas à computação. Na década de 1940, com a criação do primeiro computador moderno, eram utilizados cartões perfurados para facilitar o processo de programação e diminuir a quantidade de erros introduzidos pelo programador. Não foi até meados de 1950 que surgiu a primeira linguagem de programação moderna: FORTRAN, criada por John Backus. Os seguintes anos foram frutíferos, vieram acompanhados de duas novas linguagens de programação: LISP - John McCarthy e COBOL - Grace Hopper.

No começo, todas as linguagens de programação somente permitiam a criação de programas monolíticos e careciam de recursos que facilitassem sua utilização. Somente no fim da década de 1970 que foram estabelecidos os principais paradigmas de programação conhecidos hoje em dia: imperativo, funcional e lógico. Durante estes anos, surgiu o termo "Programação Estruturada", que visava restringir o uso de desvios incondicionais (GoTo) (ORGANICK; FORSYTHE; PLUMMER, 2014).

Em 1980, foi criada C++, que combinava orientação a objetos e programação de sistemas, também foi introduzida uma mudança de pensamento na concepção de linguagens de programação, junto com o movimento RISC em arquitetura de computadores, despertou-se maior interesse no uso de compiladores para linguagens de alto nível.

Com a chegada da internet, surgiram as linguagens de scripting, que não são evolução direta de nenhuma linguagem já estabelecida anteriormente, e que foram concebidas com novas sintaxes e novas funções (CERUZZI, 1998).

## **Ambiente e Linguagem de Programação**

Como o objetivo do trabalho é aproximar os alunos das linguagens de programação modernas, optamos por escolher uma linguagem que seja amplamente usada na atualidade, também sabemos que ela deve ser multi paradigma, já que devemos implementar soluções utilizando dois paradigmas diferentes. Pelos motivos citados previamente, escolhemos **C++**.

## **Problema Abordado**

A intenção inicial foi a de resolver um problema que já fosse conhecido pelos integrantes do grupo e que despertasse o interesse de todos, portanto escolhemos **Tower Defence**.

## 2 A LINGUAGEM C++

A linguagem de programação C++ foi criada por Bjarne Stroustrup nos anos de 1980, vindo a ter sua padronização ISO apenas 18 anos depois em 1998. Ela é uma linguagem compilada multi-paradigma, com suporte ao modelo imperativo, ao orientado a objetos, ao genérico, entre outros. Por causa disso, é de uso amplo entre as linguagens comerciais e acadêmicas. Pode-se dizer que C++ é uma linguagem que tem um domínio amplo de aplicação, já que pode ser utilizada para programar micro-controladores(Baixo Nivel), configurar interfaces de rede e desenvolver soluções para os mais variados problemas. Atualmente é utilizada para desenvolver compiladores, editores de texto, jogos, ferramentas de programação, bibliotecas entre outros(?).

### Algumas características do C++

**Operadores:** O C++ possui todo o conjunto de operadores do C, além de alguns implementados apenas no C++, que dizem respeito à conversão entre tipos, os quais que podem ser `const_cast`, `static_cast`, `dynamic_cast` e `reinterpret_cast`. Além disso, a linguagem possui sobrecarga de operadores, permitindo que um mesmo operador tenha mais do que 1 significado dependendo do contexto em que é utilizado.

**Pré-Processador:** antes da compilação propriamente dita, o C++ passa pelo seu pré-processador, gerando modificações léxicas que servem como entrada para a compilação.

**Objetos:** O C++ tem suporte aos conceitos de orientação à objetos, permitindo a criação de classes que apresentam quatro características desses conceitos: abstração, encapsulamento, herança e polimorfismo. O encapsulamento permite proteger atributos e métodos do objeto, dessa forma é possível que outros trechos do programa tenham acesso apenas aos métodos de interface com a classe. A herança de classes permite que uma classe herde atributos e métodos de outra, podendo ser relacionado com a ideia de classes mãe e filha. O polimorfismo trata da capacidade de se utilizar um operador ou método de diferentes maneiras, facilitando a estendibilidade da classe.

**Tratamento de Exceções:** Erros podem ser tratados pelo sistema, permitindo que a aplicação se recupere de algum erro sem travar ou ter de ser fechada.

**Espaço de Nomes:** Permite uma melhor organização das bibliotecas, de forma que cada uma pode criar o seu próprio espaço de nomes para que não existam conflitos.

Maiores informações podem ser encontradas em "International Standard ISO/IEC 14882:2017(E) – Programming Language C++"

### **3 TOWER DEFENCE**

É um estilo de jogo de estratégia que consiste em defender uma determinada entidade de inimigos. No nosso jogo, a entidade em questão é uma torre que se encontra no centro da tela. Esta torre possui uma certa quantidade de vida, velocidade de ataque, penetração de armadura, dano e alcance de ataque. Tais características podem ser melhoradas e outras habilidades podem ser adquiridas por meio de compras com a unidade monetária do jogo, obtida matando os inimigos.

#### **Objetivo do jogo**

Defender a sua torre de ondas progressivamente maiores de inimigos progressivamente mais fortes.

#### **Os inimigos**

Eles têm como objetivo atacar a torre até que sua vida chegue a 0 pontos, surgem de pontos aleatórios nas bordas da tela e vão em direção a torre. Eles possuem atributos definidos pelo nível do jogo, como: velocidade de ataque e de movimento, poder de ataque, quantidade de vida e de defesa. Existem 3 classes de inimigos: a classe "Soldier" se trata de um soldado que anda a pé e possui apenas armas de curto alcance. Ele vai em direção ao centro da tela e somente danifica a torre ao chegar nela. A classe "Horseman" se comporta de maneira semelhante ao soldier, porém possui mais defesa, dano de ataque e velocidade de movimento. A classe "Archer" é a que mais se diferencia das outras pois consegue atacar a torre de longas distâncias, tendo em suas características algo que as outras classes não têm, a distância de ataque, que indica a distância da qual o inimigo deve estar da torre para poder atacá-la.

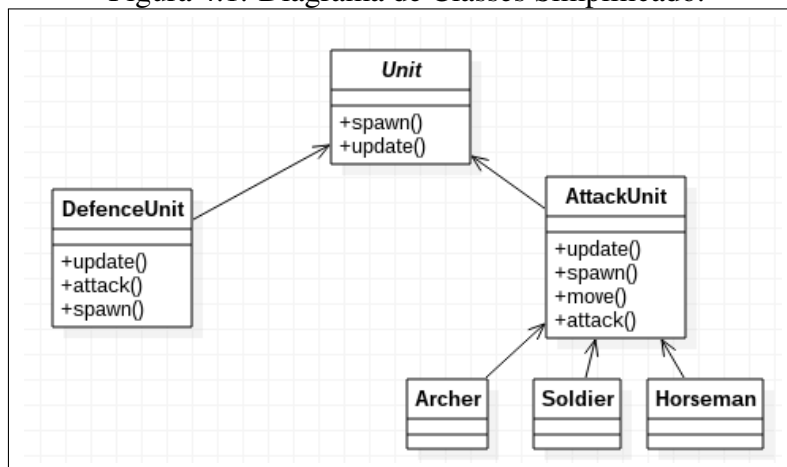
## 4 IMPLEMENTAÇÃO ORIENTADA A OBJETOS

Na linguagem C++, o paradigma orientado a objetos é o mais conhecido, logo é o mais utilizado entre os programadores. Serão introduzidos os conceitos principais abordados neste trabalho para depois realizar uma breve discussão sobre quando e onde devem ser utilizados.

### Classes

O propósito principal da programação em C++ é a acrescentar orientação a objetos à linguagem C. Classes são a característica principal da linguagem C++ que suporta orientação a objetos, também são chamadas de tipos definidos pelo usuário. A diferença das estruturas de dados presentes em C, as classes combinam a representação dos dados com métodos associados, formando assim uma unidade compacta.

Figura 4.1: Diagrama de Classes Simplificado.



### Classes Abstratas

As classes abstratas são um tipo de classes que agem como expressões de conceitos gerais das quais classes mais específicas podem ser derivadas. Não está permitido instanciar objetos de classes abstratas. Uma classe derivada de uma classe abstrata deve implementar o método virtual puro ou também será considerada uma classe abstrata.

Métodos virtuais puros são declarados da seguinte forma:

- `virtual void pureVirtualFunction() = 0`



A classe **Unit** é uma classe abstrata já que define metodos virtuais puros que são implementados nas classes derivadas correspondentes( AttackUnit e DefenceUnit).

## Herança

Herança é fundamental na programação orientada a objetos, já que fornece meios para promover a extensibilidade do código, reutilização e maior coerência lógica no modelo de implementação. As classes que são usadas para derivação são chamadas de classes base de uma classe derivada específica. Na herança a classe derivada contém os membros da classe base mais os novos membros que sejam adicionados na declaração da classe derivada. Herança é declarada da seguinte forma:

- class Derived :[virtual] [access-specifier] Base {

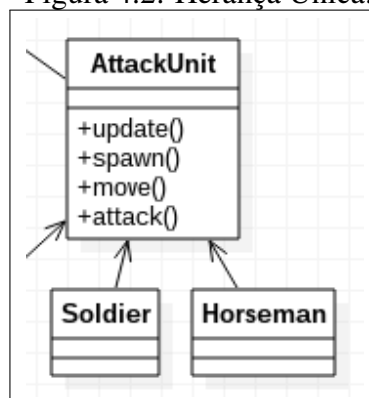
// Member List

};

## Herança Unica

As classes tem apenas uma classe base, gerando assim uma árvore de derivação.

Figura 4.2: Herança Unica.

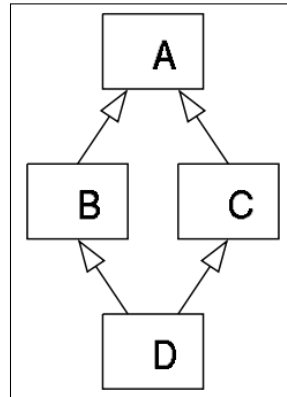


## Herança Múltipla

As classes podem herdar de mais de uma classe base, gerando assim um grafo de derivação. Herança múltipla apresenta diversos problemas de implementação:

- **Problema do Diamante:** Uma classe D herda de duas classes B e C. B e C herdam de A. Os atributos de A serão duplicados em D gerando assim conflitos e ambiguidades no momento do acesso.

Figura 4.3: Problema do Diamante.



Uma solução para este problema é utilizar herança virtual nas classes B e C.

A planificação inicial foi feita de modo a evitar o uso de Herança Multipla.

## Polimorfismo

Polimorfismo é a capacidade de objetos se comportarem de forma diferenciada em face de suas características ou do ambiente ao qual estejam submetidos, mesmo quando executando ação que detenha, semânticamente, a mesma designação.

O polimorfismo está fortemente conectado ao sistema de tipos. Já que é o sistema de tipos que define se este mecanismo é realmente implementável ou não. Uma tipagem dinâmica permite obter todos os benefícios que o polimorfismo traz assim como as desvantagens em relação aa eficiência.

## Polimorfismo Adhoc

Este tipo de polimorfismo atua somente sobre abstrações de controle, também conhecido como polimorfismo de aparência já que não faz reuso.

Tipos de polimorfismo Adhoc

- **Coersão:** Conversão implícita de tipos, pode ser de alargamento ou de estreitamento (perda de dados). Este tipo de polimorfismo aumenta a redigibilidade mas diminui

a confiabilidade do sistema. Este tipo de polimorfismo encontra-se em quase todos os projetos, já que é muito simples de utilizar, ao realizar uma operação binária entre dois operandos de tipos diferentes mas compatíveis por coersão. Ex: Somar um int com um float.

- Sobrecarga: Redefinição de itens já existentes. O mesmo operador comporta-se de maneiras diferentes dependendo dos operandos. C++ oferece total versatilidade neste sentido, já que permite sobreescrever grande quantidade dos operadores. No projeto podemos encontrar este tipo de polimorfismo nas funções `spawn()` e `update()`.

## **Polimorfismo Universal**

Também conhecido como polimorfismo verdadeiro, aplica-se o mesmo código sobre elementos de tipos diversos. Permite a programação genérica. Tipos de polimorfismo Universal

- Inclusão: Elemento de um subtipo também é um elemento do supertipo, logo os métodos da superclasse podem ser chamados com elementos da subclasse. A vinculação tardia permite este tipo de polimorfismo que também pode ser atingido via Downcasting ou Upcasting. Este tipo de polimorfismo pode encontrar-se no método `render()` da classe `Unit`, este método é chamado pelos objetos das classes `AttackUnit` e `DefenceUnit`, ambas são classes derivadas de `Unit`.
- Paramétrico: Permite o desenvolvimento de algoritmos genéricos, aumentando assim a reusabilidade de código. A linguagem C++ oferece diversas estruturas genéricas, chamadas de templates. O nome paramétrico tem relação direta a que tipos dentro da função são parametrizáveis. É importante destacar que o compilador gerará código diferente para cada tipo do algoritmo genérico que seja instanciado. Podemos encontrar este tipo de polimorfismo nas listas que contém as diferentes unidades do jogo. Utilizamos a classe `std::vector<type>` do próprio C++. Os templates são uma forma pouco poderosa de reusabilidade de código, já que em linguagens puramente orientadas a objetos existe maior quantidade de mecanismos que permitem maior liberdade relacionada aos tipos.

## Encapsulamento

O encapsulamento consiste em isolar atributos e métodos afim de protegê-los de qualquer uso indevido, isto, junto com a devida documentação, favorece em muito a reusabilidade do código, uma vez que facilita sua correta instancição.

Na nossa implementação, fizemos com que quase, senão todos, os atributos das classes fossem protegidos, isto permite que eventuais classes que herdem destas ainda consigam utilizá-los. Aliando isso aos métodos de acesso (*getters* e *setters*), temos uma camada que torna mais difícil o assinalamento de dados inválidos ou inesperados a variáveis internas, assim tornando o código mais robusto e seguro.

No nosso código, o trecho onde essa propriedade se torna mais visível é na classe *Unit*, a qual possui dados sobre todas as unidades de ataque e de defesa presentes no jogo. Aqui, todas as variáveis são protegidas para que nenhum contexto de fora possa fazer alterações diretas. Isto se tornou importante não só para manter a integridade dos dados, mas também porque todos os atributos de ataque de defesa devem se manter consistentes com seus níveis já que, apenas assim, o sistema de *upgrades* pode funcionar corretamente. Portanto, todas as alterações a esses atributos só podem ser feitas por meio dos métodos que levam em consideração os seus valores base e seus níveis atuais.

## Delegates

Delegates consistem em uma maneira de generalizar tarefas que possuem diferentes rotinas e/ou entradas para um mesmo fim. No nosso código, encontramos dificuldades em implementar Delegates como normalmente são feitos em C++, com classes ou structs específicas para este fim. Portanto, tentamos utilizar características desse modelo em algumas funções específicas. Os melhores exemplos disso a serem citados na nossa implementação desta etapa do trabalho estão na classe *Unit* e dizem respeito ao sistema de *upgrades*.

Esse sistema necessita de três informações básicas: o nível de um atributo, o seu valor base (inicial) e um coeficiente de aumento. Então juntamos estas três informações em três chamadas que realizam as seguintes operações sobre todos os atributos: consultar um valor, verificar e incrementar seu nível atual. Estas funções possuem em comum um parâmetro que consiste em uma *enum* que define o nome de um dado (vida, armadura, dano de ataque, número de alvos, velocidade de ataque e alcance de ataque). Desta forma,

cada função pode realizar seis ações diferentes, sendo uma para cada atributo.

## Espaço de Nomes

Espaço de nomes é o nome dado para o do contexto de onde vem uma variável ou função de um programa, comumente utilizado para resolver problemas de ambiguidade de um programa complexo. O espaço de nomes(“namespace”) serve para permitir que 2 identificadores idênticos possam existir em diferentes contextos do sistema, um exemplo disto é o sistema de arquivos do computador, podemos pensar no namespace como sendo uma pasta, dentro dela, não podemos ter mais do que 1 arquivo com o mesmo nome, porém, podemos ter vários arquivos com o mesmo nome, desde que eles existam apenas em pastas(namespaces) diferentes, o mesmo vale para a linguagem C++, podemos ter várias ou métodos com o mesmo nome, desde que cada uma esteja dentro do seu namespace.

O uso no namespace é bem simples de ser entendido, uma vez definido, basta utilizar o nome escolhido seguido de “::” antes do nome da variável ou método que você pretende identificar.

Figura 4.4: Espaço de nomes exemplo.

```

1 namespace exemplo1
2 {
3     int var = 1;
4 }
5
6 namespace exemplo2
7 {
8     int var = 2;
9 }
10
11 int main()
12 {
13     printf("Exemplo 1: %d, Exemplo 2: %d", exemplo1::var, exemplo2::var);
14 }
15
16
17

```

Um exemplo de uso desse recurso para a programação orientada a objetos é poder definir uma classe em um arquivo .h e implementar seus métodos em outro arquivo.cpp, permitindo assim que o programador forneça acesso aos “Headers” das funções sem ter que mostrar suas implementações, e também, tornando o programa mais “legível” e organizado. Na primeira foto, temos o arquivo AttackUnit.h onde ocorre a declaração da classe AttackUnit e a definição de alguns métodos como seu destrutor: “ AttackUnit()” e “update()”.

Na segunda fotos, temos o arquivo AttackUnit.cpp, onde temos a implementação dos métodos já definidos no arquivo .h. Aqui podemos ver o uso do namespace “AttackUnit” para indicar ao compilador que os métodos ali implementados pertencem ao contexto

da classe `AttackUnit`. Esse tipo de estruturação do código nos permite definir diferentes classe com os mesmos métodos em diferentes arquivos.h e implementar todos esses métodos no mesmo arquivo.cpp, pois o namespace resolve o problema de ambiguidade entre os nomes dos métodos.

Figura 4.5: Espaço de nomes exemplo.

```

1 #ifndef ATTACK_UNIT_H
2 #define ATTACK_UNIT_H
3 #include <stdio.h>
4 #include "Unit.h"
5
6 class AttackUnit : public Unit{
7
8 protected:
9     uint _speed, _directionX, _directionY;
10    int _reward;
11
12 public:
13    ~AttackUnit();
14    int update(Unit* target);
15    int getReward();
16
17 protected:
18    void spawn(int screenWidth, int screenHeight);
19    void move(float distanceToTower, float distance, int directionX, int directionY);
20    int attack(Unit* target);
21 };
22 #endif
23

```

Figura 4.6: Espaço de nomes.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include "AttackUnit.h"
5
6 AttackUnit::~~AttackUnit()
7 {
8     SDL_DestroyTexture(_visualTex);
9 }
10
11 int AttackUnit::update(Unit* target)
12 {
13     int elapsedTime;
14     float distanceToMove, distanceToTower;
15     int defenceTowerX, defenceTowerY;
16     int rangedAttackDamage = 0;
17

```

## Construtores e Destrutores

Construtores e Destrutores são outra característica da programação orientada a objetos, eles são métodos que são chamados, como o próprio nome já diz, quando uma classe é “construída” e “destruída”, explicando melhor, quando uma nova instância da classe é criada, logo após é chamado seu construtor, que normalmente é usado para a inicialização de valores do objeto, bem como alocação de memória e chama de outros métodos necessários para aquele objeto. O seu destrutor, por outro lado, é chamado imediatamente antes desta instância deixar de existir, liberando a memória que foi alocada pelo objeto e efetuando alguma outra operação antes que o objeto seja destruído.

Um construtor é facilmente identificado por carregar consigo o mesmo nome da sua classe, e seu destrutor, da mesma maneira, apenas tendo o símbolo “~” precedendo sua definição. Como o construtor da classe é um método como qualquer outro, ele pode

ser sobrecarregado, permitindo que uma classe tenha diferentes construtores para cada situação desejada no código. Abaixo temos um exemplo da sintaxe de uma classe com 2 construtores e seu destrutor.

Figura 4.7: Construtores e destrutores.

```

1 class Exemplo{
2     Exemplo(); // Este é um construtor da classe Exemplo
3     Exemplo(int i, char* str); // Este é outro construtor da classe Exemplo
4     ~Exemplo(); // Este é o destrutor da classe Exemplo
5 }
6
7

```

Um exemplo de uso de construtores é inicializar os valores de um objeto para que o mesmo seja usado na execução do problema. Abaixo temos a classe Archer, que representa um dos inimigos da torre dentro do jogo TowerDefence. Aqui, podemos identificar com facilidade quem é o construtor e quem é seu destrutor devido ao til( ). No construtor do exemplo abaixo podemos ver que vários valores do objeto são inicializados, para que o mesmo possa ser usado no futuro, incluísse alguns utilizando da chamada de outros métodos do programa. No destrutor podemos ver que ele efetua uma liberação de memória que foi alocada durante a execução do programa e que não será mais necessária agora que o objeto está sendo destruído.

Figura 4.8: Construtores e destrutores.

```

1 #include "Archer.h"
2 #include <SDL2/SDL_image.h>
3
4 Archer::Archer()
5 {
6     //Determina os valores base de atributos
7     _meleeDamage = 0;
8     _baseHealth = 20;
9     _baseArmour = 0;
10    _baseRangedDamage = 10;
11    _baseNumberOfTargets = 1;
12    _baseAttackDelay = 500;
13    _baseAttackRange = 60;
14    //Determina os valores dos atributos
15    _totalHealth = getAttributeValue(HEALTH, _healthLevel);
16    _armour = getAttributeValue(ARMOUR, _healthLevel);
17    _rangedDamage = getAttributeValue(DAMAGE, _healthLevel);
18    _numberOfTargets = getAttributeValue(TARGETS, _healthLevel);
19    _attackDelay = getAttributeValue(DELAY, _healthLevel);
20    _attackRange = getAttributeValue(RANGE, _healthLevel);
21    _currentHealth = _totalHealth;
22    _reward = 15;
23    _width = 10;
24    _height = 10;
25    _speed = 25;
26    _unitType = ARCHER;
27 }
28
29
30 Archer::~Archer()
31 {
32     SDL_DestroyTexture(_visualTex);
33 }
34

```

## Análise Crítica

<b>Crítérios</b>	<b>Nota</b>	<b>Justificativas</b>
<b>Simplicidade</b>	6	Pode se tornar bastante complexa uma vez que é necessário administrar ponteiros para listas e alocação de memória, como foi feito na classe Game para controlar todos os inimigos presentes no jogo.
<b>Ortogonalidade</b>	6	Não permite muita extensão das funcionalidades dos operadores e tipos básicos, assim sendo necessárias diversas funções aparentemente básicas através de includes, como a biblioteca string.h.
<b>Expressividade</b>	5	Por ser mais baixo nível, são necessárias mais linhas de código para fazer menos, como exemplo podemos citar as várias linhas de código utilizadas para manejar as listas de unidades mostradas na tela bem como os vários cálculos utilizados para mover unidades.
<b>Adequabilidade</b>	9	
<b>Variedade de estruturas de controle</b>	10	Tem todas as estruturas de controle necessárias presentes nas linguagens de programação atuais.
<b>Mecanismos de definição de tipos</b>	10	Permite a declaração de structs, enums bem como a redefinição de nomes com typedefs. Esses mecanismos foram muito utilizados na classe Unit, com a definição de enums para todos os tipos de unidades, por exemplo.
<b>Suporte a abstração de dados e de processos</b>	10	Permite a criação de classes, que, por si só, permitem um nível muito abrangente de abstração de dados.
<b>Modelo de tipos</b>	10	Possui um rígido controle de uso dos diferentes tipos disponibilizados, o que torna o desenvolvimento mais trabalhoso porém compensa uma vez que não dá muita margem de erro para o programador, tornando o código mais robusto.



<b>Crítérios</b>	<b>Nota</b>	<b>Justificativas</b>
<b>Portabilidade</b>	10	É uma linguagem muito portátil, possui compiladores para uma vasta quantidade de arquiteturas de processadores e sistemas operacionais.
<b>Reusabilidade</b>	10	É bastante reusável pois permite fácil importação das mais variadas bibliotecas.
<b>Suporte e documentação</b>	10	Microsoft oferece uma ampla documentação, rica em exemplos e conteúdo confiável, fora toda a documentação produzida por usuários em fóruns na internet.
<b>Tamanho de código</b>	7	Permite o uso de templates para a criação de algoritmos e estruturas genéricas. Facilita a programação, mas o código é replicado quando passa pelo compilador.
<b>Generalidade</b>	7	a
<b>Eficiência e custo</b>	10	A diferença da maioria das linguagens Orientadas a objetos, é uma linguagem compilada, o que outorga um alto desempenho, também permite utilizar vinculação tardia para permitir polimorfismo.

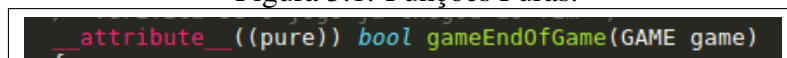
## 5 IMPLEMENTAÇÃO FUNCIONAL

Devido às intensas e constantes mudanças que ocorrem com a tecnologia, C++ foi adaptada para converter-se numa linguagem multiparadigma, adicionando suporte e estruturas do paradigma funcional, mas sem perder as principais características que a definem.

### Funções Puras

As funções puras sempre retornam o mesmo resultado para o mesmo argumento independentemente do ambiente e momento de execução. Para considerar uma função como pura, esta não deve ter efeitos colaterais no programa como modificar uma variável global, imprimir no I/O ou transmitir um pacote pela rede. Em poucas palavras, uma função pura depende exclusivamente de seus parâmetros. Funções que tem efeito colateral são chamadas de funções impuras, entre elas temos `rand()`, `time()`, `printf()` entre outras. Se uma função é marcada como pura, o compilador pode aplicar otimizações especiais como otimizações de laço além de eliminação de subexpressões, melhorando o tempo de execução do programa. Este recurso foi utilizado nas funções `gameEndOfGame` e `gameStartNewGame`. Na sintaxe de C++, devemos marcar a função como pura como pode ser observado na figura 5.

Figura 5.1: Funções Puras.



```
__attribute__((pure)) bool gameEndOfGame(GAME game)
```

Também utilizamos objetos imutáveis, especificamente listas, isto pode ser observado nas funções que adicionam novos elementos às listas, já que sempre que a lista é modificada, uma nova lista é criada, modificada e devolvida ao usuário.

### Funções Lambda

As funções lambda são o elemento principal do Cálculo lambda, criado por Alonzo Church na década de 1930 e são fundamentais para a computação teórica. A sintaxe para definir funções Lambda em C++ tem suas peculiaridades.

```
[]() mutable throw() -> int { }
```

- **Clausula de Captura:** Indica quais variáveis do escopo do chamador serão acessíveis

dentro do corpo da função lambda, esta captura pode ser dada por valor ou por referência

- Lista de parâmetros: Como em uma função normal, os parâmetros que serão utilizados.
- Indicador de mutabilidade: Permite modificar as variáveis que são passadas por valor.
- Exception: Tratamento de exceções
- Tipo de retorno: Se não é especificado é automaticamente inferido pelo compilador.
- Corpo da função

As funções lambda foram usadas como parâmetro das funções de primeira ordem na função `gameUpdate()`

Figura 5.2: Funções Lambda.

```
auto updateProjectile = [](PROJECTILE& proj)
{
    if(!proj._isDead)
    {
        projectileUpdate(&proj);
    }
};
```

### Funções como elementos de 1ª ordem

Consiste na manipulação de funções como se fossem variáveis no paradigma imperativo, o que permite atribuí-las a variáveis, listas, *structs*, etc.. No nosso projeto, utilizamos este mecanismo como uma alternativa às funções virtuais da etapa anterior. Para tanto, criamos uma *struct* com diversos campos que descrevem atributos de unidades de ataque e defesa, sendo quatro desses campos usados para funções, as quais podem variar entre os diferentes tipos de unidades. Assim, ao inicializar uma nova entidade, podemos definir qual método deve ser chamado para atualizar o comportamento, inicializar as variáveis de posição (X e Y), renderizar e recuperar a vida.

## Funções de ordem superior

Funções de ordem superior são rotinas que usam funções como valor de retorno ou como parâmetro.

O uso mais típico desse mecanismo é a substituição de laços iterativos por uma única chamada que tem como parâmetros uma lista de elementos e uma função a ser aplicada a cada nodo. Esse mesmo caso foi utilizado no nosso trabalho nas funções *gameUpate* e *gameRender*, no arquivo *Game.cpp*, para aplicar as funções de atualização do comportamento e de renderização para todas as entidades presentes no jogo. Para tanto, utilizamos a função *for\_each* definida na biblioteca *iostream*.

Além disso, usamos uma função própria de ordem superior como uma maneira de criar unidades inimigas (arqueiros, cavaleiros e soldados, criados pela rotina *createAttackUnit*). Essa função recebe como seu único parâmetro uma outra que será utilizada para a inicialização dos atributos específicos da unidade escolhida. Assim, conseguimos separar a definição das variáveis padrões para qualquer unidade daquela para o tipo específico de inimigo escolhido.

## Currying

Permite que diversas funções, de um único parâmetro cada, sejam criadas a partir de uma outra com diversos parâmetros, afim de atingir uma computação equivalente à desta. A nós, isto não especialmente útil pois conseguimos resolver todos os nossos problemas utilizando outros mecanismos funcionais, entretanto incluímos um exemplo deste mecanismo na função *gameHandleEvents*, dentro do arquivo *Game.cpp*, apenas para demonstrar como seria implementado em C++ e em que tipo de caso poderia ser utilizado.

## Recursao

A recursão é a ocorrência de uma chamada de função dentro dela mesma, essa funcionalidade é comumente utilizada em situações onde temos um caso base que define o problema e passos de recursão que ocorrem sobre este caso base, um exemplo disto é a sequência de Fibonacci onde o caso 0 e o caso 1 são definidos como casos base, e seus casos subsequentes são o passo recursivo dos dois casos anteriores, se fossemos criar um

laço for para resolver este problema, a solução seria muito mais complexa em comparação com a solução recursiva, assim, a recursão pode ser útil para resolver problemas complexos de maneira mais simples e rápida.

Voltando ao exemplo da sequência de Fibonacci, para utilizarmos a recursão é muito simples, basta efetuarmos a chamada da função que estamos criando dentro dela mesma, sempre atendendo para a ocorrência de parada. O uso da recursão requer alguns cuidados na hora de programarmos, pois temos que atender à condição de parada, bem como à sequência de chamadas para garantir que tudo irá funcionar corretamente.

Figura 5.3: Recursao.

```

1  int fibonacci(int num)
2  {
3      if(num==1 || num==2)
4          return 1;
5      else
6          return fibonacci(num-1) + fibonacci(num-2);
7  }

```

Outro exemplo de uso da recursão é a criação de listas encadeadas, onde cada passo de recursão gera um elemento e appenda o mesmo em uma lista, e logo após, retorna essa lista para o passo de recursão anterior, que cria o seu elemento, appenda ele, e retorna novamente a lista, ao final da recursão teremos a lista com todos os elementos dentro ela prontos para serem utilizados.

Figura 5.4: Recursao.

```

303 std::vector<UNIT> createUnitList(int i, UNIT_TYPE unit, int size, GAME* game)
304 {
305     std::vector<UNIT> list;
306     if(i < size)
307     {
308         UNIT newUnit;
309         switch(unit)
310         {
311             case ARCHER:
312                 newUnit = createAttackUnit(ARCHER);
313                 break;
314             case HORSEMAN:
315                 newUnit = createAttackUnit(HORSEMAN);
316                 break;
317             case SOLDIER:
318                 newUnit = createAttackUnit(SOLDIER);
319                 break;
320             default:
321                 fprintf(stderr, "Error, DEFENCE should not be used in this function\n");
322         }
323         newUnit.spawnFunction(&newUnit, game->_screenWidth, game->_screenHeight);
324         i++;
325         list = createUnitList(i, unit, size, game);
326         list.push_back(newUnit);
327     }
328     return list;
329 }
330 }

```

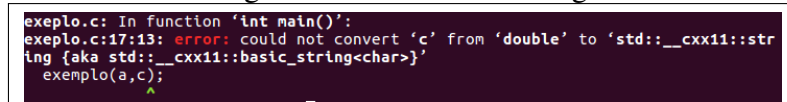
## Pattern Matching

O casamento de padrões, ou, pattern matching, é o ato de checar se uma certa sequência de valores combina ou coincide com um padrão desejado, permitindo assim

que diferentes decisões sejam tomadas de acordo com o padrão encontrado.

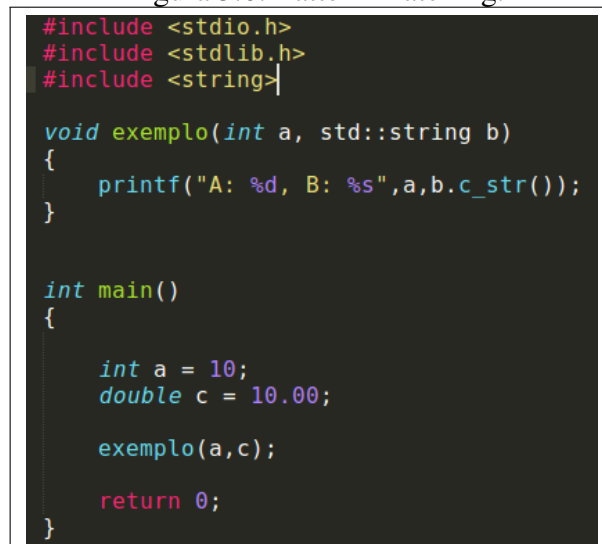
Um exemplo do uso de pattern matching é a verificação da entrada de parâmetros de uma função, o uso de tal técnica pelo compilador permite o mesmo avaliar e decidir se os valores passados por parâmetros para uma função realmente satisfazem o que aquela função está esperando, caso contrário, o mesmo pode avisar o programador sobre o erro para que ele seja corrigido.

Figura 5.5: Pattern Matching.



```
exemplo.c: In function 'int main()':
exemplo.c:17:13: error: could not convert 'c' from 'double' to 'std::__cxx11::string {aka std::__cxx11::basic_string<char>}'
    exemplo(a,c);
               ^
```

Figura 5.6: Pattern Matching.



```
#include <stdio.h>
#include <stdlib.h>
#include <string>

void exemplo(int a, std::string b)
{
    printf("A: %d, B: %s",a,b.c_str());
}

int main()
{
    int a = 10;
    double c = 10.00;

    exemplo(a,c);

    return 0;
}
```

Nas imagens acima podemos ver que o compilador acusa o erro de que não pôde converter a variável double c em String, acusando assim um erro de combinação pois a função exemplo espera que sejam passados para ela um int e uma string.

Outro exemplo mais explícito do qual o programador pode tirar proveito é o Overload ou sobrecarga de funções, onde, a partir do pattern matching, o compilador decide qual instância da função usar de acordo com os parâmetros que foram passados para ele, permitindo assim que o programador utilize um único nome de função para diferentes tipos de dados.

Figura 5.7: Pattern Matching.

```
205 void render(UNIT& U, GAME* game)
206 {
207     if(U._currentHealth > 0){
208         if(!U.renderFunction(&U, game->_renderer, game->_screenWidth, game->_screenHeight))
209         {
210             // Tenta novamente
211             U.renderFunction(&U, game->_renderer, game->_screenWidth, game->_screenHeight);
212         }
213     }
214 }
215
216 void render(PROJECTILE& P, GAME* game)
217 {
218     if(!P._isDead){
219         if(!projectileRender(&P, game->_renderer, game->_screenWidth, game->_screenHeight))
220         {
221             // Tenta novamente
222             projectileRender(&P, game->_renderer, game->_screenWidth, game->_screenHeight);
223         }
224     }
225 }
226
```

## 6 CONCLUSÃO

Após aprofundar nossos conhecimentos da linguagem C++ e solucionar um mesmo problema utilizando duas abordagens totalmente diferentes, podemos concluir que cada paradigma tem um domínio específico e, embora cada um tenha seus pontos fortes e fracos, acreditamos que uma combinação de ambos fornece a solução mais adequada.

A solução orientada a objetos foi responsável por um código mais limpo e claro em termos de estrutura devido à hierarquia de classes simples e intuitiva e à facilidade na definição de *namespaces*. Entretanto, também resultou em mais trabalho para obedecer as regras impostas por estas facilidades estruturais no momento de definir a interação entre as classes e a posição de todas as funções e atributos.

O modelo funcional nos permitiu reduzir significativamente certos trechos de código, especialmente aqueles em que empregamos funções de ordem superior aliadas a funções lambda (por exemplo, as rotinas *render* e *update* no arquivo *Game.cpp*) e torná-los mais facilmente compreensíveis por meio de funções como *for\_each*. Isto porque foram quebradas algumas das barreiras entre funções e variáveis. O que, por outro lado, exigiu maior complexidade sintática e um conhecimento mais aprofundado das especificidades da linguagem escolhida, conhecimento esse que não seria diretamente transferido para outras linguagens.

A linguagem de programação escolhida facilitou muito a implementação das duas etapas do trabalho pois é muito popular e ampla, o que é um fator determinante para a vasta quantidade de material disponível. Entretanto, a usabilidade do C++ para o paradigma orientado a objetos é muito superior àquela para o funcional. Isto porque é uma linguagem nascida para a orientação a objetos que sofreu alterações posteriores à sua criação que permitiram e facilitaram o uso em múltiplos paradigmas.

Portanto, não surpreendentemente, pensamos ser possível criar um código melhor aliando perspectivas de diferentes paradigmas de programação. Isto, entretanto, não é uma dica de ouro que permite a todos programar melhor instantaneamente, mas sim serve como mais uma demonstração e que são necessárias profundas pesquisas e testes para melhorar as habilidades de codificação. Além disso, o conhecimento geral teórico não é suficiente, é necessário que cada linguagem a ser empregada seja conhecida e experimentada afim de atingir o melhor equilíbrio para trabalho em questão.



## REFERÊNCIAS

CERUZZI, P. **A History of Modern Computing** (Cambridge, MA & London. [S.l.]: MIT Press, 1998.

ORGANICK, E. I.; FORSYTHE, A. I.; PLUMMER, R. P. **Programming language structures**. [S.l.]: Academic Press, 2014.