Seminário Sobre a Linguagem de Programação C++

Augusto Stambassi Duarte
Davi Cândido de Almeida
João Pedro Torres
Lucas Carneiro Nassau Malta

[1]Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
Belo Horizonte, MG – Brazil

Resumo. *O C++ destaca-se por sua profunda herança histórica e influência nas linguagens modernas, ao evoluir da linguagem C e incorporar conceitos da programação orientada a objetos, como classes e herança. Seu suporte ao multiparadigma permite combinar estilos procedurais, orientados a objetos e funcionais, tornando-o altamente versátil. Essa flexibilidade ampliou seu uso em sistemas críticos, jogos e aplicações de alto desempenho. Além disso, a adoção da programação orientada a objetos revolucionou a forma de estruturar problemas computacionais, promovendo modularidade, reutilização de código e maior abstração na construção de softwares complexo*

Sumário

1. Histórico da Linguagem

Este tópico tem como objetivo discutir a história da linguagem de programação C++, apresentando sua cronologia histórica, ou seja, as sequências de fatos que deram origem ao que conhecemos hoje como C++, bem como sua genealogia, aprofundando nas relações e transformações que a envolveram durante esse processo e destacando o impacto de cada um desses fatos em sua evolução e em seu uso até os dias atuais. Para isso, foi realizada uma revisão bibliográfica em diferentes modalidades de obras, como livros e artigos escritos pelo próprio autor da linguagem e fontes pertencentes à literatura cinzenta, como blogs, vídeos e relatórios periódicos.

Antes de citar o c++ se faz necessário contextualizar o seu passado e as diversas linguagens anteriores, cada uma contribuindo com aspectos técnicos e conceituais fundamentais. O FORTRAN I (1957), como primeira linguagem compilada de alto nível, introduziu recursos que permitiram maior produtividade em relação à programação em Assembly. Embora limitado a uma única máquina, foi essencial para demonstrar o potencial das linguagens de alto nível. Pouco depois, o ALGOL 58 e sua evolução, o ALGOL 60, introduziram conceitos revolucionários como a descrição formal de linguagens, tipagem mais rica, estruturação em blocos e procedimentos recursivos — características que ecoariam tanto em C quanto em C++. Apesar de não terem se popularizado como o FORTRAN, essas linguagens influenciaram profundamente o design das linguagens modernas, especialmente em termos de sintaxe e estrutura algorítmica.[Sebesta 2000]

Entre essas influências, destaca-se o SIMULA 67, uma extensão do ALGOL 60 que introduziu o conceito de classes e herança para simulação de sistemas — marcos que inauguraram a programação orientada a objetos. Tais conceitos foram diretamente incorporados por Bjarne Stroustrup ao desenvolver o C++, na tentativa de unir a eficiência do C com a capacidade de abstração de linguagens como SIMULA. Já o Smalltalk (1980), embora mais voltado à interação com usuários não-programadores, reforçou o paradigma orientado a objetos de forma mais radical, ao tratar tudo como objetos e promover a comunicação entre eles por meio de envio de mensagens — uma ideia que influenciou a modelagem e design de sistemas orientados a objetos que mais tarde também seriam possíveis no C++.[Sebesta 2000]

Diante disso, a história da linguagem C++ tem início nos Bell Telephone Laboratories (Bell Labs), centro de pesquisa fundado pela AT&T em 1925, reconhecido por suas contribuições científicas e tecnológicas nas áreas de telecomunicações, eletrônica e informática. Na década de 1970, Dennis Ritchie, pesquisador do Bell Labs, esteve envolvido no desenvolvimento de um novo sistema operacional, o Unix, projetado para ser multitarefa, multiusuário e altamente adaptável. No entanto, a criação de um sistema com tais características exigia programas compactos, eficientes e com controle preciso sobre o hardware, o que se tornava dificultoso com o uso de linguagens Assembly. Por serem linguagens de baixo nível, com implementações específicas para cada arquitetura de processador, a programação em Assembly tornava-se complexa, trabalhosa e pouco portátil. Essas limitações motivaram a busca por linguagens de programação mais produtivas, capazes de manter o desempenho e ao mesmo tempo oferecer maior abstração e flexibilidade no desenvolvimento de software [Stroustrup 2002, Fróes and Weber 2023, Santiago 2024].

Nesse cenário de busca por maior produtividade e flexibilidade na criação de sis-

temas, Dennis Ritchie iniciou, em colaboração com Ken Thompson, o desenvolvimento de uma nova linguagem de programação de mais alto nível em relação às linguagens existentes à época. O resultado desse esforço foi a criação da linguagem B, em 1969, projetada inicialmente para o PDP-11, um minicomputador da Digital Equipment Corporation (DEC), conhecido por ter sido utilizado no primeiro *arcade* operado por moedas, com o jogo Galaxy. Com o tempo, a linguagem passou por aprimoramentos importantes, como a introdução de tipos de dados distintos para variáveis, o que ampliou seu potencial de aplicação. Essas evoluções permitiram que a linguagem B fosse utilizada também em *mainframes* da Honeywell e em determinados sistemas embarcados ao longo da década de 1970.

Embora a linguagem B não tenha sido amplamente adotada para seu propósito inicial — a implementação do sistema operacional Unix —, seu desenvolvimento desempenhou um papel fundamental na evolução das linguagens de programação. Ela serviu como base direta para a criação da linguagem C, que posteriormente se consolidaria como uma das linguagens mais influentes e amplamente utilizadas na história da computação [Stroustrup 2002, Stroustrup 1994].

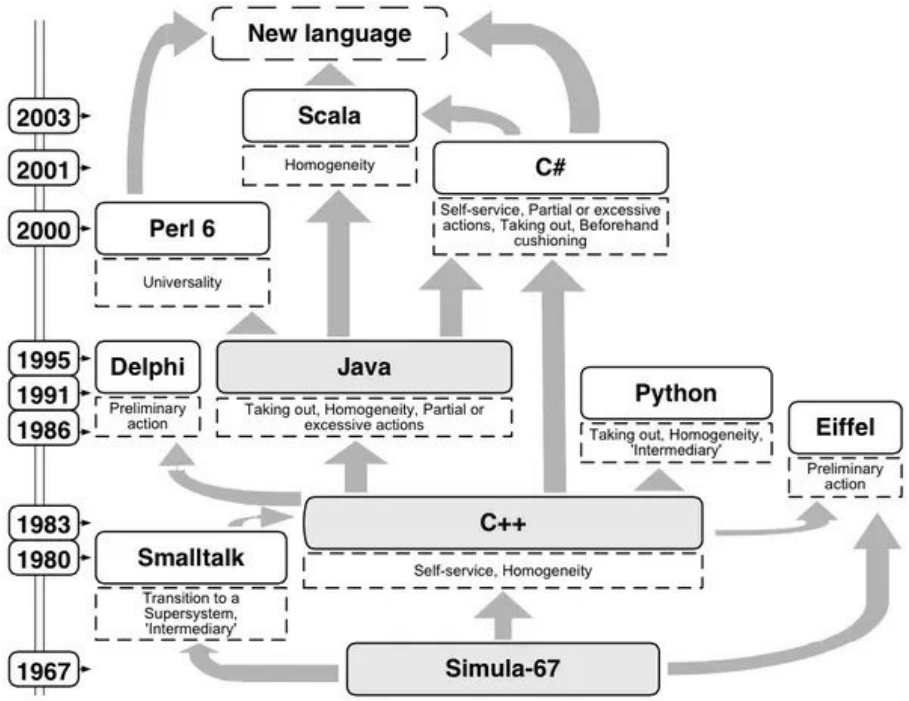
Em 1978, foi publicado o livro *The C Programming Language*, escrito por Brian Kernighan e Dennis Ritchie, obra que se tornaria amplamente conhecida como K&R C, em referência aos sobrenomes dos autores. Este livro teve papel fundamental na consolidação e padronização da linguagem C, apresentando sua estrutura formal ao público e contribuindo para sua ampla disseminação fora do ambiente restrito do Bell Labs. A publicação marcou o início da popularização da linguagem, que rapidamente se tornou uma das mais influentes da história da computação. Em 1988, foi lançada a segunda edição da obra, refletindo o padrão ANSI C, estabelecido pelo American National Standards Institute (ANSI). O impacto do livro e da linguagem é duradouro: a linguagem C continua a influenciar gerações de programadores e serviu como base conceitual e sintática para o desenvolvimento de várias outras linguagens, incluindo o próprio C++ [Stroustrup 1994, Fróes and Weber 2023].

Apesar da ampla adoção da linguagem C, o cientista dinamarquês Bjarne Stroustrup, também integrante do Bell Labs, identificou a necessidade de uma linguagem que permitisse o desenvolvimento de sistemas complexos com maior capacidade de abstração, sem abrir mão do alto desempenho e do controle preciso de recursos — características essenciais em domínios como sistemas operacionais e sistemas embarcados. Com esse objetivo, Stroustrup iniciou, no início da década de 1980, o desenvolvimento de uma extensão da linguagem C que incorporasse conceitos da programação orientada a objetos, inspirando-se em linguagens como Simula 67 e Smalltalk-80, que já exploravam tais paradigmas. O resultado foi a criação do C with Classes (“C com classes”), um superset de C que introduzia recursos como classes, encapsulamento e construtores, mantendo compatibilidade com a sintaxe e a eficiência da linguagem original [Stroustrup 2002, Stroustrup 1994, Fróes and Weber 2023]. Ainda nesse contexto, Stroustrup comenta sobre a criação dessa linguagem como uma possibilidade de facilitar a criação de códigos mais próximos ao entendimento dos desenvolvedores:

> “C++ foi desenvolvido para que meus amigos e eu não tivéssemos que programar em Assembly, C ou outra linguagem moderna. Seu propósito era escrever programas de forma mais fácil e agradável para o programador.”

[Stroustrup 2002]

No início da década de 1980, mais precisamente entre os anos de 1979 e 1982, ocorreu o lançamento oficial da linguagem de programação C++, anteriormente conhecida como C with Classes. A mudança de nome não foi apenas nominal, mas simbólica: o uso do operador de incremento ”++”, herdado da linguagem C, indicava que o C++ representava uma evolução natural e incremental de sua predecessora. A principal preocupação de Bjarne Stroustrup, seu criador, era desenvolver uma linguagem prática e eficiente, mais orientada à utilidade do que a seguir uma filosofia de projeto rígida. Em 1985, foi publicada a primeira edição da obra The C++ Programming Language, escrita pelo próprio Stroustrup, com o intuito de promover e formalizar o uso da nova linguagem. Assim como o clássico de Ritchie e Kernighan havia feito com a linguagem C, o livro de Stroustrup consolidou o C++ como uma linguagem versátil e poderosa, especialmente por seu suporte à orientação a objetos. A obra teve impacto significativo na difusão desse paradigma, influenciando diretamente o desenvolvimento de linguagens posteriores, como Java, C#, Python e Objective-C. A Figura 1 ilustra a genealogia da linguagem C++, destacando sua origem e as influências fundamentais em sua evolução.[Stroustrup 1994, Stroustrup 2002, Fróes and Weber 2023]



Fonte: [Berdonosov and Zhivotova 2014]

Alguns anos após a publicação de The C++ Programming Language, em 1990 foi lançado o livro The Annotated C++ Reference Manual (ARM), obra de fundamental importância na história do C++. Este manual desempenhou papel crucial ao se tornar o primeiro documento formal que padronizou a linguagem, servindo como base oficial para o processo de normalização internacional conduzido pelos comitês ISO e ANSI. Tal padronização foi essencial para mitigar os problemas de fragmentação que a linguagem vinha enfrentando até então, consolidando o C++ como uma linguagem madura e

consistente. O ARM, portanto, estabeleceu os fundamentos para o padrão C++98, a primeira versão oficial e amplamente reconhecida da linguagem. Atualmente, o padrão mais recente é o C++23 (2023), que integra o ciclo de atualizações trienais adotado desde 2012.[Ellis and Stroustrup 1990, Stroustrup 1994, Lab C++ 2023]

Historicamente, diversos compiladores marcaram a evolução da linguagem C++. O primeiro deles foi o Cfront, desenvolvido por Bjarne Stroustrup no início dos anos 1980, que funcionava como um tradutor de código C++ para C, possibilitando que programas escritos em C++ fossem compilados por compiladores C já existentes. Essa estratégia facilitou a disseminação inicial da linguagem. Além disso, o Cfront adotou a técnica de *self-hosting*, ou seja, a capacidade de compilar seu próprio código-fonte, demonstrando a viabilidade de compiladores escritos na própria linguagem que compilam a si mesmos. Contudo, o Cfront foi descontinuado em 1993, principalmente devido à dificuldade de suportar novas funcionalidades, como o tratamento de exceções.

Outro compilador amplamente utilizado, especialmente no meio acadêmico e profissional, é o GCC (GNU Compiler Collection). Desenvolvido como parte do Projeto GNU, iniciado por Richard Stallman em 1984, o GCC destaca-se pelo suporte a múltiplas linguagens, incluindo C e C++, e sua ampla adoção em sistemas Unix e Linux. Sua importância vai além do aspecto técnico, pois foi um motor central na disseminação do software livre e de código aberto, mantendo-se ativo e evoluindo continuamente. Já o Clang, compilador baseado na infraestrutura LLVM, desenvolvido por volta dos anos 2000, destaca-se por sua modularidade e pela clareza das mensagens de erro, que facilitam a depuração e o desenvolvimento. O Clang tem sido adotado como compilador padrão em diversas plataformas, como o macOS, e utilizado por grandes empresas, como Google e Apple, em projetos de grande escala, incluindo o desenvolvimento do navegador Google Chrome. Segundo a pesquisa [C++ Stories 2022], aproximadamente 70,9% dos desenvolvedores utilizavam o GCC em 2022, enquanto 46,1% faziam uso do Clang.

2. Características Principais e Paradigmas de Programação

A linguagem C++ é uma linguagem de programação de propósito geral, compilada, multiparadigma e de alto desempenho. Criada por Bjarne Stroustrup na década de 1980 como uma extensão da linguagem C, o C++ foi desenvolvido com o objetivo de combinar a eficiência e o controle de baixo nível do C com mecanismos de abstração mais modernos, herdados da programação orientada a objetos [Stroustrup 2013]

2.1. Principais características da linguagem C++

C++ é uma linguagem de propósito geral cujas características marcantes incluem desempenho elevado, portabilidade e suporte a múltiplos paradigmas. Seu acesso direto à memória e o controle explícito sobre alocação e desalocação de recursos possibilitam otimizações de baixo nível, tornando-a ideal para aplicações que exigem alta eficiência, como sistemas embarcados, jogos, bancos de dados e motores gráficos. Além disso, a ampla disponibilidade de compiladores compatíveis com diversas plataformas — como GCC, Clang e MSVC — garante sua portabilidade entre diferentes arquiteturas e sistemas operacionais. Por fim, seu suporte a múltiplos paradigmas, como o imperativo procedural, o orientado a objetos e o funcional declarativo, oferece ao desenvolvedor flexibilidade para escolher o estilo de programação mais apropriado a cada contexto [Stroustrup 2013].

2.2. Paradigmas de Programação de C++

2.2.1. Paradigma Imperativo Procedural

O paradigma imperativo procedural fundamenta-se na emissão de comandos que alteram o estado interno do programa ao longo da execução. Em C++, esse modelo é nativamente suportado, pois a linguagem deriva diretamente do C, que adota essa abordagem de forma predominante. Sua estrutura baseia-se no uso de variáveis e instruções de controle condicional e iterativa, como *if*, *for* e *while*, que organizam a lógica do programa de maneira sequencial e controlada [Stroustrup 2013].

Entre as principais características do paradigma imperativo, destaca-se o fluxo de controle explícito, no qual o código é estruturado como uma sequência ordenada de instruções que definem, passo a passo, o comportamento do programa. Essa abordagem permite que o programador tenha controle direto sobre a execução, utilizando estruturas como laços, condicionais e desvios para guiar o processamento. Outra característica fundamental é o uso de funções, que viabiliza a modularização do código por meio da divisão em blocos lógicos reutilizáveis. Isso facilita a organização do programa, promove o reuso de lógica comum e melhora a legibilidade e manutenção. Além disso, o paradigma imperativo é baseado no conceito de estado, ou seja, os programas operam sobre variáveis que armazenam valores e são modificadas ao longo da execução. Essa mutabilidade permite representar e acompanhar a evolução do sistema em tempo de execução, mas também pode introduzir complexidade na depuração e no rastreamento de efeitos colaterais [Sebesta 2000].

Um exemplo de código que apresenta o paradigma imperativo procedural pode ser observado no Código 2.2.1.

Código 2.2.1: Trecho de código com o paradigma imperativo procedural

```cpp
#include <iostream>

int main() {
    int x = 10;
    x = x + 5;
    std::cout << x; // imprime 15
    return 0;
}
```

2.2.2. Paradigma Imperativo Orientado a Objetos (OOP)

A orientação a objetos constitui um dos principais pilares da linguagem C++, introduzida por Bjarne Stroustrup como uma evolução em relação ao C, ao incorporar mecanismos de abstração originários de linguagens como Simula 67 e Smalltalk-80 [Sebesta 2000]. Essa abordagem promove a organização do código em torno de objetos, entidades que encapsulam tanto o estado interno (atributos) quanto os comportamentos associados (métodos), favorecendo a modelagem de sistemas complexos de forma mais intuitiva e modular [Sebesta 2000].

O C++ fornece suporte robusto aos principais princípios da orientação a objetos: herança, polimorfismo, encapsulamento e abstração. A herança possibilita a extensão e reutilização de características de classes preexistentes, promovendo economia de código e coesão hierárquica. O encapsulamento restringe o acesso direto ao estado interno do objeto, preservando a integridade dos dados e permitindo controle sobre sua modificação. Já o polimorfismo viabiliza a definição de comportamentos distintos para métodos com a mesma assinatura, adaptando sua execução ao tipo de objeto em tempo de compilação ou de execução. Esses mecanismos, combinados, tornam o paradigma orientado a objetos um recurso poderoso para o desenvolvimento de sistemas complexos, reutilizáveis e escaláveis [Stroustrup 2013, Sebesta 2000].

Um exemplo de código que apresenta o paradigma imperativo orientado a objetos pode ser observado no Código 2.2.2.

Código 2.2.2: Trecho de código com o paradigma imperativo orientado a objetos

```cpp
#include <iostream>
#include <string>
using namespace std;

class Pessoa {
    string nome;
public:
    Pessoa(const string& n) : nome(n) {}
    void apresentar() const {
        cout << "áOl, meu nome é " << nome << "." << endl;
    }
};

int main() {
    Pessoa p("Maria");
    p.apresentar();
```

```
17      return 0;
18  }
```

2.2.3. Paradigma Declarativo Funcional

Embora o C++ não seja uma linguagem funcional no sentido estrito, a partir do padrão C++11 publicado em 2011 [Foundation 2011] passou a incorporar uma série de recursos que viabilizam a adoção de práticas associadas ao paradigma funcional. Dentre esses recursos, destacam-se as funções lambda, que permitem a criação de funções anônimas de forma concisa; o uso de *const* para promover imutabilidade, restringindo a modificação de dados após sua definição; e o suporte a funções de ordem superior, como *std::transform* e *std::accumulate*, que possibilitam a manipulação de coleções de dados de maneira declarativa e composicional [Stroustrup 2013, Foundation 2011].

A utilização do paradigma funcional em C++ favorece a produção de um código mais modular, previsível e menos suscetível a efeitos colaterais, ao restringir a dependência de estados mutáveis e ao enfatizar o uso de funções puras. Dentre as características centrais dessa abordagem, destacam-se: funções puras, que operam exclusivamente sobre seus parâmetros e não modificam variáveis externas; expressões lambda, que oferecem uma sintaxe enxuta para definir comportamentos locais e parametrizáveis; e a ênfase na imutabilidade, que desencoraja alterações de estado, promovendo maior segurança e confiabilidade na execução do programa [Foundation 2011].

Código 2.2.3: Trecho de código com o paradigma declarativo funcional

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <numeric> // para std::accumulate
5
6  using namespace std;
7
8  int main() {
9      vector<int> numeros = {1, 2, 3, 4, 5};
10
11     // Funcao lambda pura para dobrar um numero
12     auto dobrar = [](int x) {
13         return x * 2;
14     };
15
16     // Aplicar std::transform (funcao de primeira ordem) para mapear a
           funcao 'dobrar' sobre o vetor
17     vector<int> dobrados(numeros.size());
18     transform(numeros.begin(), numeros.end(), dobrados.begin(), dobrar)
           ;
19
20     // Aplicar std::accumulate (funcao de primeira ordem) para somar os
           valores do vetor transformado
21     int soma = accumulate(dobrados.begin(), dobrados.end(), 0);
22
23     // Exibir os resultados
24     cout << "Valores dobrados: ";
```

```
25      for (int n : dobrados)
26          cout << n << " ";
27
28      cout << "\nSoma dos valores dobrados: " << soma << endl;
29
30      return 0;
31  }
```

2.3. Vantagens da Programação Multi-Paradigma

A linguagem C++ oferece suporte a múltiplos paradigmas de programação, permitindo que o desenvolvedor selecione a abordagem mais apropriada conforme a natureza do problema. Além disso, é possível combinar diferentes paradigmas dentro de um mesmo projeto, promovendo maior flexibilidade e expressividade no desenvolvimento de software.

Essa característica favorece a manutenção e reutilização de código, uma vez que cada parte do sistema pode ser escrita com a técnica mais adequada ao seu propósito. Também permite a otimização localizada — como o uso de construções de baixo nível ou algoritmos eficientes — sem comprometer a clareza ou a organização estrutural proporcionada por abstrações de alto nível.

Por exemplo, é comum estruturar os principais componentes do sistema com orientação a objetos, aproveitando encapsulamento e modularidade, enquanto funções lambda ou algoritmos funcionais são empregados para realizar operações pontuais sobre coleções de dados de forma concisa e eficiente. Essa integração harmônica entre paradigmas torna o C++ uma linguagem poderosa e versátil, adequada tanto para aplicações de sistema quanto para projetos de alto nível.

3. Linguagens relacionadas

Como explorado o C++ é uma linguagem de grande uso no mercado, reconhecido pelo seu desempenho e eficiência, o que a torna a preferida para programação em nível de sistema, desenvolvimento de jogos, aplicações em tempo real e muito mais. Dessa maneira, vale explorar a relação de C++ com outras linguagens, para compreender melhor o seu uso, em quais situações ela é mais indicada e também explorar técnicas de integração com outras linguagens, a fim de aproveitar recursos exclusivos de cada uma.

3.1. Linguagens que influenciaram o C++

3.1.1. Linguagem C

Ao falar das linguagens que influenciaram o C++, é necessário fazer a retomada da linguagem C. Afinal, C++ surgiu como uma extensão de C que buscava incorporar abstrações de alto nível, preservando o desempenho e o acesso a baixo nível que definiam C. Assim, C++ não foi concebido como um substituto, mas como um superconjunto, um superconjunto que pudesse se integrar perfeitamente às bases de código C, às ferramentas e às expectativas dos programadores existentes.

Embora C++, tenha herdado a sintaxe, o modelo de memória e o pipeline de compilação de C, divergiu em recursos como verificação de tipos robusta, classes, sobrecarga de funções, modelos e exceções, todos projetados com sobrecarga mínima de tempo de execução para respeitar a filosofia de eficiência de C.

Um exemplo marcante, que pode ser fornecido é à filosofia de "confiar no programador". Assim como C, C++ permite construções potencialmente inseguras se mal utilizadas, como manipulação direta de memória e aritmética de ponteiros. No entanto, C++ estende essa confiança oferecendo alternativas mais seguras e de nível superior, como a vector classe na Biblioteca Padrão de Modelos (STL), que oferece verificação de limites quando desejado, sem impô-la universalmente.

Código 3.1.1: Trecho de código em C (Potencialmente inseguro)

```c
#include <stdio.h>

void print_array(int *arr, int size) {
    // ERRO: acesso fora dos limites
    for (int i = 0; i <= size; i++) {
        printf("%d\n", arr[i]);
    }
}

int main() {
    int values[3] = {10, 20, 30};
    print_array(values, 3);
    return 0;
}
```

Código 3.1.1: Trecho de código em C++ (com abstração mais segura)

```cpp
#include <iostream>
#include <vector>
```

```
3
4  void print_vector(const std::vector<int>& vec) {
5      // ERRO: ípossvel çãexceo
6      for (size_t i = 0; i <= vec.size(); ++i) {
7          // .at() faz çãverificao de limites
8          std::cout << vec.at(i) << std::endl;
9      }
10 }
11
12 int main() {
13     std::vector<int> values = {10, 20, 30};
14     try {
15         print_vector(values);
16     } catch (const std::out_of_range& e) {
17         std::cerr << "Acesso fora dos limites: "
18         << e.what() << std::endl;
19     }
20     return 0;
21 }
```

Outra abordagem da influência do C em C++ é que Stroustrup enfatizou que, no espírito da linguagem C, a biblioteca padrão da linguagem C++ deveria ser implementável na própria linguagem. Isso levou a um ecossistema em que abstrações como contêineres, algoritmos e iteradores são fornecidas como construções extensíveis pelo usuário, em vez de sintaxes pré-definidas. Essas funcionalidades refletem a ênfase do C++ em mecanismos de abstração reutilizáveis, ao contrário da preferência de C por soluções mais diretas e construídas a partir de tipos primitivos.

Com o tempo, as linguagens evoluíram em direções diferentes. O C99, por exemplo, introduziu recursos como arrays de tamanho variável (VLAs), tipos complexos e o tipo _Bool, focando principalmente em computação numérica tradicional. Já o C++98 e versões posteriores optaram por evitar novas primitivas sempre que possível, preferindo enriquecer a biblioteca padrão com templates e classes, oferecendo maior generalidade e segurança sem sacrificar desempenho.

Essas decisões de design resultaram em divergências que impactam a interoperabilidade entre as linguagens. Enquanto o C continua sendo preferido em contextos onde simplicidade e portabilidade são cruciais (como sistemas embarcados), o C++ é mais adequado para aplicações que exigem estruturação complexa, reutilização de código e manutenção de longo prazo.

3.1.2. Linguagem Simula

Além de C, outra linguagem que teve grande influência foi a linguagem Simula, desenvolvida por Ole-Johan Dahl e Kristen Nygaard no início dos anos 1960 no Norwegian Computing Center, é considerada a primeira linguagem de programação orientada a objetos. Sua versão mais influente, Simula 67, introduziu conceitos revolucionários como classes, objetos, herança, encapsulamento e co-routines, que mais tarde se tornaram pilares da programação moderna.

Bjarne Stroustrup após trabalhar com Simula durante o seu doutorado em Cam-

bridge, apreciou a sua capacidade de abstração, porém percebeu que seu sistema era inadequado para sistemas de grande escala. A partir disso, Stroustrup idealizou sua união com a eficiência de C, em C++, unindo os pontos positivos de ambas linguagens.

Código 3.1.2: Simulação de um contador simples com classe em Simula 67

```
Begin
    Class Counter;
        Integer count;

        Procedure Increment;
        Begin
            count := count + 1;
        End;

        Procedure Print;
        Begin
            OutInt(count, 5);
            OutImage;
        End;

        Begin
            count := 0;
        End;

    CounterClass: Counter;

    Begin
        CounterClass.Increment;
        CounterClass.Increment;
        CounterClass.Print;
    End;
End;
```

Código 3.1.2: Simulação de um contador simples com classe em C++

```
#include <iostream>

class Counter {
    int count;

public:
    Counter() : count(0) {}

    void Increment() {
        count++;
    }

    void Print() const {
        std::cout << count << std::endl;
    }
};

int main() {
    Counter counter;
    counter.Increment();
```

```
21      counter.Increment();
22      counter.Print();
23      return 0;
24  }
```

3.2. Linguagens Influenciadas

3.2.1. Java e C#

Com o crescimento da linguagem C++ e o seu reconhecimento como uma ponte da programação estruturada e a orientação de objetos de alto desempenho surgiram linguagens influenciadas, com a intenção explícita de preservar os benefícios do C++, ao mesmo tempo que eliminam ou mitigam suas complexidades mais problemáticas.

Do ponto de vista sintático e conceitual, Java herdou diretamente a estrutura de declaração de classes, a organização em métodos e o uso de modificadores de acesso do C++. No entanto, foi projetada para ser mais segura e portável, eliminando mecanismos como ponteiros explícitos, destrutores manuais e herança múltipla. Em vez disso, Java introduziu o conceito de interfaces, que se tornaram uma alternativa segura e flexível para o compartilhamento de comportamento entre tipos, em contraste com a herança múltipla típica do C++

No caso do C#,, desenvolvido pela Microsoft no início dos anos 2000, a influência do C++ é igualmente evidente. C# adota uma sintaxe similar e compartilha os princípios fundamentais da programação orientada a objetos, como encapsulamento, herança e polimorfismo. No entanto, a linguagem é fortemente inspirada também pelo modelo de segurança e produtividade de Java, incorporando elementos como coleta automática de lixo, propriedades, delegados e suporte nativo a interfaces.

Embora Java e C# apresentem estruturas mais padronizadas e previsíveis, o C++ continua a oferecer maior flexibilidade, sendo frequentemente utilizado em contextos de sistemas embarcados, jogos e aplicações que exigem controle detalhado de recursos. Essa diferença de enfoque influencia diretamente a forma como os desenvolvedores estruturam seus sistemas em cada linguagem.

Código 3.2.1: Herança Múltipla em C++

```
1  #include <iostream>
2
3  class Imprimivel {
4  public:
5      virtual void imprimir() const = 0;
6  };
7
8  class Serializavel {
9  public:
10     virtual void salvar() const = 0;
11 };
12
13 // Classe que herda de duas classes base (çherana úmltipla)
14 class Documento : public Imprimivel, public Serializavel {
15 public:
16     void imprimir() const override {
```

```
17          std::cout << "Imprimindo documento...\n";
18      }
19
20      void salvar() const override {
21          std::cout << "Salvando documento...\n";
22      }
23  };
24
25  int main() {
26      Documento doc;
27      doc.imprimir();
28      doc.salvar();
29      return 0;
30  }
```

Código 3.2.1: Herança Múltipla implementada com Interfaces em Java

```
1   interface Imprimivel {
2       void imprimir();
3   }
4
5   interface Serializavel {
6       void salvar();
7   }
8
9   // Classe implementa úmltiplas interfaces (sem çherana úmltipla de
        classes)
10  public class Documento implements Imprimivel, Serializavel {
11
12      @Override
13      public void imprimir() {
14          System.out.println("Imprimindo documento...");
15      }
16
17      @Override
18      public void salvar() {
19          System.out.println("Salvando documento...");
20      }
21
22      public static void main(String[] args) {
23          Documento doc = new Documento();
24          doc.imprimir();
25          doc.salvar();
26      }
27  }
```

3.2.2. Rust

O fato de C++ oferecer flexibilidade e baixo nível de abstração, trazem um custo: a responsabilidade manual por segurança de memória. Nesse contexto podemos trazer o relatório da Google (Rebert & Kern, 2024), que mostra que cerca de 70 porcento das vulnerabilidades críticas em sistemas como Chrome, Android e servidores corporativos estão relacionadas à ausência de segurança de memória — falhas como use-after-free,

buffer overflows e null pointer dereferencing são recorrentes em grandes bases de código C++.

Nesse sentido linguagem Rust emerge como uma resposta direta às limitações históricas do C++, mantendo seus pontos fortes de desempenho e controle sobre o hardware, mas substituindo a responsabilidade do programador por um sistema de garantias formais em tempo de compilação. O núcleo dessas garantias é o "borrow checker", que impõe regras rígidas sobre propriedade, empréstimo e tempo de vida das referências, eliminando classes inteiras de bugs comuns no C++ sem necessidade de coleta de lixo.

Segundo o relatório, Rust é hoje a única linguagem de produção amplamente adotada que fornece segurança temporal e espacial de memória sem depender de mecanismos de runtime como garbage collection ou contadores de referência universais. Isso contrasta com C++, onde a gestão da memória continua sendo tarefa do desenvolvedor e fontes frequentes de falhas difíceis de depurar e mitigar. Além disso, Rust também garante segurança contra condições de corrida (data races) em código seguro, oferecendo o que a comunidade chama de "fearless concurrency".

Embora Rust não seja um "sucessor direto" do C++ do ponto de vista da sintaxe ou interoperabilidade perfeita, ele representa uma evolução conceitual dos princípios que motivaram o surgimento do C++: controle, eficiência e abstração. Porém, Rust avança com uma proposta centrada em segurança por construção (secure by design), impondo por padrão o uso de um subconjunto seguro da linguagem, permitindo o uso de blocos de unsafe apenas quando absolutamente necessário e mediante revisão especializada, como apontado no artigo.

Portanto, pode-se afirmar que Rust, apesar de romper com a compatibilidade sintática com C++, preserva o espírito de linguagem de sistemas de alto desempenho, ao mesmo tempo que soluciona, com inovação, problemas que o C++ acumulou ao longo de sua evolução.

Código 3.2.2: Use-After-Free em C++ (comportamento indefinido)

```
#include <iostream>

void problema() {
    int* ptr = new int(42);
    // libera a ómemria
    delete ptr;
    /* uso óaps çãliberao →
    comportamento indefinido */
    std::cout << *ptr;
}

int main() {
    problema();
    return 0;
}
```

Código 3.3.1: Use-After-Free Erro em tempo de compilação em Rust

```
fn problema() {
    let ptr = Box::new(42);
```

```
3       // ownership de ptr é movida para raw
4       let raw = ptr;
5       // println!("{}", ptr);
6       // erro: 'ptr' foi movido, ão pode
7       // mais ser usado
8       println!("{}", raw);
9   }
10
11  fn main() {
12      problema();
13  }
```

3.3. Linguagens Similares

3.3.1. Eiffel

Diversas linguagens compartilham fundamentos conceituais ou estruturais com o C++, seja por herança direta, por similaridade sintática ou por foco em desempenho e abstração. Entre elas, destacam-se D, Objective-C e Eiffel. Com destaque a seguir para a linguagem Eiffel.

C++ e Eiffel são linguagens orientadas a objetos com tipagem estática, mas representam propostas distintas de projeto. Enquanto o C++ prioriza desempenho e flexibilidade como linguagem multiparadigma, o Eiffel foca na correção formal e segurança por meio de um modelo puramente orientado a objetos.

Um dos principais diferenciais está no suporte à programação por contrato. Eiffel integra nativamente pré-condições, pós-condições e invariantes, promovendo verificação formal do comportamento de objetos. Já o C++ carece desse suporte direto, delegando tal responsabilidade ao programador via práticas manuais.

No gerenciamento de memória, o C++ oferece controle total, mas assume riscos como vazamentos e acessos inválidos. Eiffel adota coleta de lixo automática, oferecendo maior segurança, ainda que com menor controle de alocação. Ambas suportam herança múltipla, mas enquanto o C++ utiliza herança virtual e escopo explícito para resolução de conflitos, o Eiffel recorre a cláusulas como rename, redefine e select, que tornam os conflitos mais visíveis e gerenciáveis.

Por fim, os mecanismos de visibilidade também diferem: C++ usa modificadores (public, private, friend), ao passo que Eiffel emprega cláusulas de exportação, definindo com precisão quais classes podem acessar determinadas funcionalidades.

Assim, C++ e Eiffel refletem visões complementares: o primeiro voltado à performance e flexibilidade, o segundo à robustez e previsibilidade estrutural.

Código 3.3.1: Exemplo em C++ (sem suporte nativo a contratos)

```
1   #include <iostream>
2   #include <cassert>
3
4   class ContaBancaria {
5   private:
6       double saldo;
7
```

```
8  public:
9      ContaBancaria() : saldo(0.0) {}
10
11     void depositar(double valor) {
12         assert(valor >= 0); // épr-çãcondio manual
13         double saldo_antes = saldo;
14         saldo += valor;
15         // óps-çãcondio manual
16         assert(saldo == saldo_antes + valor);
17     }
18
19     double obterSaldo() const {
20         return saldo;
21     }
22 };
23
24 int main() {
25     ContaBancaria* conta = new ContaBancaria();
26     conta->depositar(100.0);
27     std::cout << "Saldo: " << conta->obterSaldo() << std::endl;
28     delete conta; // gerenciamento manual de ómemria
29 }
```

Código 3.3.1: Exemplo em Eiffel (com contratos e coleta de lixo)

```
1  class
2      CONTA_BANCARIA
3
4  create
5      make
6
7  feature
8      saldo: REAL
9
10     make
11         do
12             saldo := 0.0
13         end
14
15     depositar (valor: REAL)
16         require
17             valor_nao_negativo: valor >= 0
18         do
19             saldo := saldo + valor
20         ensure
21             saldo_atualizado: saldo = old saldo + valor
22         end
23 end
```

3.4. Linguagens 'Opostas'

3.4.1. Python

Ao longo da evolução da computação, diversas linguagens surgiram com propostas contrastantes às do C++. Entre essas, destacam-se Ruby, Haskell, JavaScript e especialmente

Python, que adota uma filosofia radicalmente distinta e o qual vamos abordar nesse momento.

C++ e Python representam visões contrastantes sobre o desenvolvimento de software. Enquanto o C++ oferece controle detalhado de recursos, alto desempenho e múltiplos paradigmas, Python prioriza legibilidade, simplicidade e produtividade.

No C++, o programador é responsável por gerenciar memória, tipos e desempenho, o que permite otimizações finas, mas exige maior cuidado e complexidade. Python, por sua vez, adota tipagem dinâmica, coleta automática de lixo e uma sintaxe minimalista, reduzindo barreiras para o desenvolvimento, mesmo que com menor previsibilidade de execução.

Além disso, C++ confia no programador para garantir a correção do código, enquanto Python favorece soluções padronizadas e seguras por design. Essas diferenças tornam C++ preferido em sistemas críticos e de tempo real, enquanto Python domina áreas como ciência de dados, automação e prototipagem rápida.

4. Atividade prática

Como parte das atividades deste seminário, foi realizada uma demonstração do ambiente gráfico da *Unreal Engine 5*, um motor amplamente utilizado na indústria de desenvolvimento de jogos. O objetivo dessa atividade foi ilustrar parte do potencial da linguagem C++ discutido ao longo da apresentação, evidenciando por que ela é tão empregada na construção de motores gráficos.

A instalação da *Unreal Engine 5* pode ser realizada conforme descrito na Seção 5.2. Para reproduzir o modelo utilizado na apresentação, basta fazer o download do arquivo compactado (ZIP) disponível no repositório oficial do projeto no *GitHub*.

Durante a apresentação, foi utilizado um menu interativo, desenvolvido em C++ dentro do ambiente da *Unreal Engine*, conforme ilustrado no Código 4. Ressalta-se que a lógica de programação na *Unreal Engine* depende fortemente da orientação a objetos, utilizando-se de classes e bibliotecas específicas do próprio motor gráfico. Dessa forma, a classe responsável pelos slides da apresentação, por exemplo, foi implementada como uma subclasse de `UUserWidget`, como demonstrado no Código 4, o que assegura a compatibilidade com a interface do motor.

Código 4: Código para controle da apresentação

```
#include "SlideshowWidget.h"
#include "Components/Image.h"
#include "Components/Button.h"
#include "Engine/Texture2D.h"
#include "Slate/SlateBrushAsset.h"
#include "Kismet/GameplayStatics.h"
#include "Input/Reply.h"
#include "Input/Events.h"
#include "GameFramework/PlayerController.h"
#include "Blueprint/UserWidget.h"

int32 USlideshowWidget::LastIndexBeforeLeaving = -1;

bool USlideshowWidget::Initialize()
{
    Super::Initialize();
    SetKeyboardFocus();


    if (NextButton)
    {
        NextButton->OnClicked.AddDynamic(this, &USlideshowWidget::
            OnNextClicked);
    }

    if (LastIndexBeforeLeaving >= 0)
    {
        CurrentIndex = LastIndexBeforeLeaving + 1;

        // Prote contra overflow
        if (CurrentIndex >= ImageList.Num())
        {
            CurrentIndex = ImageList.Num() - 1;
```

```
33          }
34
35          // Reseta para que da pra vez comece do 0
36          LastIndexBeforeLeaving = -1;
37      }
38      else
39      {
40          CurrentIndex = 0; // ou outro valor padr     }
41
42      UpdateImage();
43      return true;
44  }
45
46  void USlideshowWidget::OnNextClicked()
47  {
48      if (ImageList.Num() == 0) return;
49      if (CurrentIndex != 30) {
50          CurrentIndex = (CurrentIndex + 1) % ImageList.Num();
51          UpdateImage();
52      }
53  }
54
55  void USlideshowWidget::OnPreviousClicked()
56  {
57      if (ImageList.Num() == 0) return;
58
59      CurrentIndex = (CurrentIndex - 1) % ImageList.Num();
60      UpdateImage();
61
62  }
63
64  void USlideshowWidget::UpdateImage()
65  {
66      if (!SlideImage || ImageList.Num() == 0) return;
67
68      UTexture2D* CurrentTexture = ImageList[CurrentIndex];
69
70      if (CurrentTexture)
71      {
72          FSlateBrush NewBrush;
73          NewBrush.SetResourceObject(CurrentTexture);
74          NewBrush.ImageSize = FVector2D(1920 , 1080); // ou o tamanho
                real da textura
75          SlideImage->SetBrush(NewBrush);
76      }
77  }
78
79  FReply USlideshowWidget::NativeOnKeyDown(const FGeometry& InGeometry,
        const FKeyEvent& InKeyEvent)
80  {
81      UE_LOG(LogTemp, Warning, TEXT("NativeOnKeyDown foi chamado"));
82
83      const FKey PressedKey = InKeyEvent.GetKey();
84
85      if (PressedKey == EKeys::Right)
86      {
```

```
87          OnNextClicked(); // Pro slide
88          return FReply::Handled();
89      }
90      else if (PressedKey == EKeys::Left)
91      {
92          OnPreviousClicked(); // Slide anterior
93          return FReply::Handled();
94      }
95      else if (PressedKey == EKeys::Up && CurrentIndex == 30)
96      {
97          UpdateLevel(); // Sair do Mundo e ir pro 3 Pessoa
98          return FReply::Handled();
99      }
100
101     return Super::NativeOnKeyDown(InGeometry, InKeyEvent);
102 }
103
104 void USlideshowWidget::UpdateLevel()
105 {
106     LastIndexBeforeLeaving = CurrentIndex;
107
108     APlayerController* PC = UGameplayStatics::GetPlayerController(
        GetWorld(), 0);
109     if (PC)
110     {
111         FInputModeGameOnly InputMode;
112         PC->SetInputMode(InputMode);
113         PC->bShowMouseCursor = false;
114     }
115
116     UGameplayStatics::OpenLevel(this, FName("Downtown_Alley"));
117 }
```

Além disso, é necessário declarar separadamente os cabeçalhos das classes, especificando atributos, métodos e heranças. O arquivo de cabeçalho apresentado no Código 4 define os modificadores de acesso e os *macros* específicos da *Unreal Engine*, como `UFUNCTION()`, utilizado para indicar ao motor gráfico os métodos que podem ser acessados pela lógica do jogo.

Código 4: Cabeçalho da classe de controle da apresentação

```
1  #pragma once
2
3  #include "CoreMinimal.h"
4  #include "Blueprint/UserWidget.h"
5  #include "SlideshowWidget.generated.h"
6
7  UCLASS()
8  class SEMINARIO_API USlideshowWidget : public UUserWidget
9  {
10     GENERATED_BODY()
11
12 public:
13     virtual bool Initialize() override;
14
```

```
15 protected:
16     // Referia para o Image e o Button do Blueprint
17     UPROPERTY(meta = (BindWidget))
18     class UImage* SlideImage;
19
20     UPROPERTY(meta = (BindWidget))
21     class UButton* NextButton;
22
23     // Texturas que serexibidas
24     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Slideshow")
25     TArray<UTexture2D*> ImageList;
26
27     // dice atual
28     int32 CurrentIndex = 0;
29
30     // Fun chamada ao clicar
31     UFUNCTION()
32     void OnNextClicked();
33
34     UFUNCTION()
35     void OnPreviousClicked();
36
37     // Atualiza a imagem atual
38     void UpdateImage();
39
40     void UpdateLevel();
41
42     virtual FReply NativeOnKeyDown(const FGeometry& InGeometry, const
           FKeyEvent& InKeyEvent) override;
43
44     static int32 LastIndexBeforeLeaving;
45
46
47 };
```

5. Tutorial de Instalação

5.1. Instalação do Compilador G++

A instalação do compilador G++ em sistemas baseados em Linux pode ser realizada por meio do gerenciador de pacotes da respectiva distribuição. No *Arch Linux*, por exemplo, o pacote pode ser instalado com o seguinte comando:

```
sudo pacman -S gcc
```

Já em distribuições baseadas no Debian, como o *Ubuntu*, recomenda-se a instalação do pacote `build-essential`, que inclui o G++ e outros utilitários de desenvolvimento:

```
sudo apt-get install build-essential
```

No *macOS*, o compilador pode ser instalado via o *Xcode Command Line Tools* ou, alternativamente, por meio do gerenciador de pacotes `Homebrew`:

```
brew install gcc
```

Para sistemas Windows, a instalação é um pouco mais complexa. Uma opção amplamente utilizada é o pacote `MinGW`. O usuário deve realizar o download do instalador pelo *SourceForge*, descompactar os arquivos em um diretório acessível (como `C:\MinGW\bin`), criar uma pasta `bin` e adicioná-la à variável de ambiente `Path`. Isso pode ser feito acessando as configurações de "Variáveis de Ambiente" no Painel de Controle.

5.2. Instalação da Unreal Engine

Nos sistemas *Windows* e *macOS*, a instalação da Unreal Engine é realizada por meio do *Epic Games Launcher*, disponível no site oficial da Epic Games. Após a instalação do launcher, o usuário poderá realizar o download da *Unreal Engine 5* diretamente pela interface gráfica da aplicação.

No *Linux*, o processo de instalação varia conforme a distribuição. Para sistemas baseados no Ubuntu, a Epic Games fornece um arquivo compactado contendo o instalador da plataforma. Já para usuários do *Arch Linux*, é necessário vincular a conta da Epic Games ao GitHub, clonar o repositório oficial da Unreal Engine e realizar sua compilação manualmente, conforme instruções disponíveis no próprio repositório.

6. Considerações finais

A linguagem C++ permanece como uma das ferramentas mais poderosas e influentes no desenvolvimento de software de alto desempenho. Sua capacidade de **abstração com alto desempenho** permite integrar conceitos de orientação a objetos à base procedural herdada do C, viabilizando o desenvolvimento de sistemas complexos de forma estruturada e eficiente. Essa combinação única a torna a escolha ideal para aplicações críticas como engines de jogos, sistemas de banco de dados e até mesmo sistemas operacionais.

Além disso, o C++ destaca-se pela sua **portabilidade e versatilidade**. Com suporte em compiladores disponíveis para praticamente todas as plataformas modernas, o C++ permite que aplicações sejam desenvolvidas para uma ampla gama de ambientes — desde sistemas embarcados e softwares científicos até navegadores e ferramentas de produtividade. Essa característica amplia significativamente o escopo e a relevância da linguagem no cenário atual da computação.

Entretanto, essa potência vem acompanhada de uma grande **responsabilidade para o programador**. O controle direto sobre recursos como a memória — por meio de ponteiros e operações explícitas com `new` e `delete` — confere liberdade e flexibilidade, mas também impõe riscos. Problemas como vazamentos de memória e ponteiros soltos (*dangling pointers*) são comuns em projetos que não adotam boas práticas e rigor na engenharia de software. Assim, embora o C++ proporcione um controle fino sobre o comportamento do programa, ele também exige um alto nível de disciplina e conhecimento técnico.

Dessa forma, o C++ continua sendo uma linguagem relevante e robusta, especialmente indicada para aplicações onde desempenho, controle e portabilidade são fatores cruciais — desde que o programador esteja preparado para lidar com os desafios que acompanham essa liberdade.

Referências

Berdonosov, V. and Zhivotova, A. (2014). The evolution of the object-oriented programming languages. *Scholarly Notes of Komsomolsk-na-Amure State Technical University*, 2(18):35–43.

C++ Stories (2022). C++ at the end of 2022. https://www.cppstories.com/2022/cpp-status-2022/. Acesso em: 27 abr. 2025.

Ellis, M. A. and Stroustrup, B. (1990). *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc.

Foundation, S. C. (2011). C++11 overview. https://isocpp-org.translate.goog/wiki/faq/cpp11?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc. Acesso em: 28 abr. 2025.

Fróes, G. and Weber, V. (2023). C++ (a linguagem imortal de verdade) // dicionário do programador. Acesso em: 20 abr. 2025.

Lab C++ (2023). C++: História e Versões. https://labcpp.com.br/c-historia-e-versoes-2/. Acesso em: 27 abr. 2025.

Santiago, J. (2024). A origem da linguagem c++. Acesso em: 22 abr. 2025.

Sebesta, R. W. (2000). *Concepts of programming languages*. Porto Alegre : Bookman, 4th edition edition.

Stroustrup, B. (1994). *The design and evolution of C++*. Pearson Education India.

Stroustrup, B. (2002). *A linguagem de programação C++: Bjarne Stroustrup; tradução: Maria Lúcia Blanck Lisbôa, Carlos Arthur Lang Lisbôa*. Bookman.

Stroustrup, B. (2013). *The C++ programming language*. Upper Saddle River, NJ : Addison Wesley, 4th edition edition.

Apéndices

A. Augusto Stambassi Duarte

Confesso que tive uma certa indignação quando o tema sorteado foi C++, eu gostaria de algo totalmente diferente do que eu já havia estudado e visto dentro e fora da universidade. Porém, ao longo do estudo da linguagem C++, ficou evidente sua **grande versatilidade**, demonstrada pela ampla adoção em diversos setores da indústria de software ao longo de décadas. Desde sistemas embarcados e automação industrial até aplicações gráficas avançadas e sistemas financeiros, o C++ se mostrou adaptável a contextos variados, sustentando-se como uma das linguagens mais duradouras e relevantes na história da computação moderna.

Outro aspecto importante observado foi a **escalabilidade** proporcionada pela orientação a objetos (POO). Esse paradigma, incorporado ao C++ de forma poderosa, oferece mecanismos para organização modular do código, reutilização de componentes e separação de responsabilidades — fatores fundamentais para lidar com a crescente complexidade de sistemas em expansão. Grandes empresas de tecnologia, como as chamadas Big Techs, baseiam parte significativa de sua infraestrutura em sistemas escritos (ou com componentes escritos) em C++, justamente pela escalabilidade e controle que a linguagem oferece.

Durante o aprofundamento em projetos reais, como o estudo da *Unreal Engine*, foi possível compreender melhor como a orientação a objetos influencia diretamente a forma de **planejamento e estruturação** de um projeto. O uso extensivo de herança, composição e polimorfismo não apenas organiza o código de maneira mais compreensível, mas também permite que funcionalidades complexas sejam desenvolvidas, testadas e mantidas de forma mais eficaz. Essa experiência contribuiu significativamente para o entendimento de como paradigmas de programação impactam as decisões arquiteturais em projetos de larga escala.

Essas observações reforçam a importância do C++ como uma linguagem que não apenas oferece alto desempenho, mas também se adapta bem às exigências modernas de organização e manutenção de software, sendo uma ferramenta de grande valor tanto para projetos acadêmicos quanto para aplicações comerciais e industriais.

B. Davi Cândido de Almeida

B.1. Carga histórica e influência em linguagens modernas

A trajetória histórica do C++ evidencia seu papel central no desenvolvimento das linguagens modernas. Originada como uma extensão da linguagem C — cuja eficiência e proximidade com o hardware a tornaram a espinha dorsal do desenvolvimento de sistemas —, o C++ foi concebido para unir desempenho e abstração. Bjarne Stroustrup, ao idealizar o C with Classes, buscou uma solução que mantivesse o controle de recursos do C, mas que também incorporasse os conceitos da programação orientada a objetos introduzidos por linguagens como Simula 67 e Smalltalk. Essa visão resultou numa linguagem robusta, que influenciou profundamente Java, C#, Rust e outras, moldando práticas modernas como encapsulamento, herança e controle de memória manual ou automatizado.

Assim, o C++ atua como um elo entre a programação de baixo nível e a programação moderna de alto nível, tendo suas ideias disseminadas em quase todas as principais linguagens de propósito geral em uso atualmente. O uso do operador de incremento ”++”em seu nome não é apenas simbólico — representa de fato a evolução incremental e consciente das bases do C para novas fronteiras da programação.

B.2. O multiparadigma e sua expansão de utilidades da linguagem

Uma das características mais marcantes do C++ é sua natureza multiparadigma. A linguagem não se limita a uma única forma de pensar ou resolver problemas: ela permite que o programador escolha entre paradigmas imperativo procedural, orientado a objetos e, a partir do C++11, também o funcional. Essa flexibilidade amplia imensamente as possibilidades de aplicação da linguagem — desde sistemas operacionais e jogos de alto desempenho até algoritmos numéricos e manipulação de coleções com expressões lambda.

Esse suporte a múltiplos paradigmas permite, por exemplo, que sistemas complexos sejam estruturados com orientação a objetos, mas que partes críticas de desempenho sejam otimizadas com programação procedural, ou ainda que tarefas específicas sobre dados sejam tratadas com funções puras e iteradores funcionais. Essa integração confere ao C++ uma versatilidade rara entre linguagens, tornando-o adaptável a praticamente qualquer contexto de desenvolvimento, sem impor ao programador um único estilo ou abordagem.

B.3. Programação orientada a objetos e a revolução na estruturação de problemas computáveis

A introdução da programação orientada a objetos (OOP) no C++, inspirada por linguagens como Simula 67 e Smalltalk, marcou uma ruptura no modo como problemas computacionais eram abordados. Em vez de enxergar programas como uma sequência linear de instruções, a OOP propôs um modelo mais próximo da realidade, no qual o sistema é composto por ”objetos”— entidades que combinam dados e comportamentos relacionados. O C++ adotou essa visão ao mesmo tempo em que manteve a eficiência característica do C.

Os pilares da OOP — encapsulamento, herança, polimorfismo e abstração — passaram a permitir a construção de sistemas mais modulares, reutilizáveis e robustos. A capacidade de modelar problemas através de classes promoveu uma forma mais intuitiva e escalável de desenvolvimento. Essa revolução estrutural se estende até hoje, e linguagens modernas continuam a basear seus sistemas de tipos e arquiteturas em conceitos originados ou popularizados pelo C++.

C. João Pedro Torres

Após a realização deste trabalho, percebi a relevância e o impacto do C++ em diversas aplicações e também em outras linguagens de programação. Sua combinação de alto desempenho com um nível de abstração adequado permite o desenvolvimento de sistemas eficientes e relativamente fáceis de compreender. Nesse sentido, é impressionante como o C++ funciona como um verdadeiro núcleo de desempenho, atuando nos bastidores de muitos sistemas. Bibliotecas escritas em C++ são amplamente utilizadas por

linguagens como Python, Java, C#, Rust, entre outras, o que contribui significativamente para a eficiência e produtividade no desenvolvimento de software.

Além disso, chama atenção a capacidade do C++ de acompanhar as evoluções tecnológicas, mantendo-se relevante ao se integrar com linguagens mais modernas e de maior abstração. Isso demonstra sua versatilidade e sua habilidade de transcender seu próprio paradigma, coexistindo, estendendo e servindo como base para diversas outras linguagens.

D. Lucas Carneiro Nassau Malta

Ao longo dos meus estudos sobre a linguagem C++, pude reconhecer nela uma poderosa combinação entre desempenho de baixo nível e abstrações de alto nível. Essa dualidade é uma de suas maiores virtudes: o C++ possibilita que o programador tenha controle fino sobre os recursos do sistema, como alocação de memória, uso de ponteiros e estruturas de dados otimizadas, ao mesmo tempo em que oferece mecanismos robustos de abstração, como classes, herança, polimorfismo e templates. Trata-se de uma linguagem que conseguiu adaptar conceitos modernos, como a programação orientada a objetos, ao contexto do C, uma linguagem já amplamente estabelecida na indústria e na academia. Com isso, o C++ tornou-se capaz de atender a demandas que exigem não apenas performance, mas também organização, reutilização e escalabilidade do código. Como consequência, tornou-se uma linguagem ideal para a construção de sistemas extremamente complexos e eficientes, como motores gráficos, bancos de dados, navegadores e sistemas embarcados.

Outro aspecto que me chamou atenção é a portabilidade da linguagem. O C++ é utilizado em uma ampla variedade de plataformas, desde microcontroladores de baixo custo, passando por infraestruturas de servidores corporativos, até supercomputadores de alta performance. Essa flexibilidade é viabilizada tanto pelo acesso direto ao hardware quanto pela padronização formal da linguagem, estabelecida desde 1998 e atualizada periodicamente por meio do ISO C++ Committee. Tal padronização contribui para a consistência do ecossistema, garantindo que programas escritos em C++ possam ser mantidos e executados de forma confiável em diferentes ambientes e ao longo de décadas.

Por fim, gostaria de destacar o papel relevante que o C++ ocupa no cenário da programação competitiva. A linguagem é largamente utilizada em competições sobretudo devido à Standard Template Library (STL). Essa biblioteca oferece ao programador um vasto conjunto de estruturas de dados genéricas e algoritmos otimizados, que permitem escrever soluções concisas, eficientes e de fácil manutenção. A combinação entre alta produtividade — proporcionada pela STL e pela familiaridade com a sintaxe — e desempenho elevado, oriundo do controle sobre os recursos computacionais, torna o C++ uma das ferramentas mais poderosas nesse contexto.

Em suma, minha experiência com o C++ tem reforçado a percepção de que se trata de uma linguagem equilibrada entre baixo nível, alto poder de abstração, portabilidade e desempenho, sendo capaz de acompanhar tanto os desafios da engenharia de software moderna quanto os requisitos mais exigentes da computação de alto desempenho.