# C++ 14 e C++ 17

Gianluca Ciccarelli e Luan Torres

*MAC 5742-0219 Introdução à Programação Concorrente, Paralela e Distribuída*

## 1. Introdução

A linguagem de programação *C* foi publicada em 1983, e desde então vem sendo amplamente utilizada.

Mesmo já tendo boa estabilidade, e mesmo suas versões antigas sendo as mais populares, seus desenvolvedores continuam a aprimorá-la, de forma que ela continue a suprir as necessidades de sua ampla rede de usuários.

Nesta monografia serão explicadas as funcionalidades mais básicas do *C++*, sua utilização, e também as novas *features* das suas novas versões: *C++ 14* e *C++ 17*.

## 2. C++

Ao longo da década de 70 boa parte da programação era feita em *C*, mas com o passar do tempo e introdução do conceito de Orientação a Objetos uma extensão da linguagem foi criada, chamada de *"C com classes"*, que mais tarde seria a principal motivação da criação do *C++*.

Seguindo a filosofia do *C*, todo o controle da programação é feito pelo programador, sendo possível fazer a manipilação dos dados em seu nível mais baixo, o que gera algum desconforto em programadores novos ou de linguagens mais permissivas.

Sua sintaxe é consistente, descrevendo bem o que cada linha de código faz quando usada propriamente, mas que constantemente deixa coisas mais ofuscadas para quem não está familiarizado, principalmente por conta da sua verbosidade.

```
// Array de ponteiros para funcao que retorna ponteiros para chars
char *(*func[10])(int arg);
```

Quando se trata de paradigmas, a linguagem é composta por três partes que trabalham em conjunto no código, visando tanto a eficiência computacional quanto flexibilidade, como vemos a seguir.

### *2.1. Imperativa*

A parte imperativa do *C++* é o que seria o próprio *C*, sendo possível programar na linguagem como se fosse sua antecessora sem problemas.

```c
// Hello world de C em C++
#include <cstdio> // bibliotecas padrao em c++ nao levam o .h
                  // bibliotecas do C comecam com c, stdio.h -> cstdio
int main()
{
    printf("Hello World!");
    return 0;
}
```

Por conta dessa característica, notamos que nos dias atuais praticamente não encontramos (ou usamos) compiladores somente para *C*, sendo esse trabalho delegado para os compiladores *C++*.

Além disso, sendo a linguagem uma extensão, todo o controle e eficiencia do *C* é mantido e melhorado, como será visto nos próximos tópicos.

### *2.2. Orientada a Objetos*

Sendo o foco na extensão do *C*, orientação a objetos, como o nome sugere, traz o conceito do paradigma para a linguagem, introduzindo conceitos como classes, herança e polimorfismo.

A ideia de agrupar dados relacionados em uma única estrutura já existia em *C*, e em *C++* essa ideia é melhorada para agrupar os conceitos de orientação a objetos.

```c
// Struct simples em C
typedef struct
{
    int bar;
} foo;
```

A palavra reservada struct, do *C*, serve um proposito similar no *C++*, porém ao invés de simplesmente guardar um conjunto de dados relacionados, structs são classes. Com isso possuem construtores, destrutores, funções e níveis de acesso, típicos da orientação a objetos.

```c
// Struct simples em C++
struct foo
{
    int bar; // Publico por padrao
};
```

Dito isto, existe também a palavra reservada class, que cumpre exatamente o mesmo papel de uma struct, com a única diferença que o nivel de acesso padrão de uma struct é público e de uma class é privado.

```c
// Class simples em C++
class foo
{
    int bar; // Privado por padrao
};
```

Com isso, herança entre classes e o polimorfismo temos uma ideia de como orientação a objetos foi introduzida em cima do *C*, aumentando as possibilidades para os programadores.

### *2.3. Genérica*

A programação genérica consiste em programar uma função ou comportamento sem o conhecimento prévio do tipo usado. Por ser um conceito bastante abstrato, fica dificil pensar como uma linguagem compilada, de tipagem forte e com objetivo de ser usada para alta performance poderia disponibilizar esse tipo de funcionalidade.

Em *C++* esse comportamento é atingido pelo uso de *Templates*, estruturas que permitem ao programador escrever funções genéricas sem que a eficiência do programa seja comprometida.

```
// Exemplo de template
template<class T>
void swap(T &a, T &b)
{
    T c = a;
    a = b;
    b = c;
}
```

#### *2.3.1. STL*

O conceito de *Templates* é tão poderoso que existe uma biblioteca padão do *C++* totalmente baseada em programação genérica, a *Standard Template Library*, ou *STL*.

A *STL* consiste em quatro componentes principais, sendo eles *containers*, *algorithms*, *functional* e *iterators* que juntos dão ao programador flexibilidade e eficiência tanto na execução do programa quanto no seu desenvolvimento:

- *containers*: contém estruturas de dados comuns como vetores, maps e listas ligadas.
- *algorithms*: contém algorítmos comuns que operam sobre estruturas de dados, independentemente de qual seja. Exemplos são sort e search.
- *functional*: contém objetos de funções, ou seja, objetos que são utilizados como funções.
- *iterators*: contém iteradores, objetos que são capazes de percorrer uma coleção por referencia aos seus objetos internos.

## 3. Extensões

Com o crescente uso do *C++*, naturalmente novas necessidades foram nascendo, levando ao ampliamento da linguagem de formas que mudaram o jeito de programar.

Grandes bibliotecas foram feitas para facilitar a programação ou adicionar novos aspectos, sendo a Boost uma das mais famosas.

Essa biblioteca possui diversas utilizades programadas em *C++* que foram uteis para tantos programadores que acabaram indo para a biblioteca padrão da linguagem. Bons exemplos são os smart pointers *shared pointer* e *weak pointer*.

Para que uma nova funcionalidade seja adicionada, ela primeiro deve constar em um documento chamado Technical Specifications (Technical Report até 2012, sendo o TR1 quase todo implementado no *C++ 11*) que define propostas de funcionalidades e restrições que podem fazer parte de uma versão futura do *C++*.

## 4. C++ 14

### *4.1. Shared Lock*

Um novo tipo Mutex adicionado à biblioteca padrão. São comuns casos em que múltiplas *threads* precisam acessar um mesmo conjunto de dados, e em certos casos, até modificá-los. O *Shared Lock* permite que as *threads* tenham acesso irrestrito à uma variável, sem ter que travar o Mutex para acessá-la. Porém, quando uma delas precisa modificar o valor, a trava é ativada para que somente este processo tenha acesso à variável, e continua desta forma até que as modificações estejam concluídas.

```
#include <mutex>
#include <shared_mutex>
#include <thread>

class ThreadSafeCounter {
 public:
  ThreadSafeCounter() = default;

  unsigned int get() const {
    std::shared_lock<std::shared_mutex> lock(mutex_);
    return value_;
  }

  void increment() {
    std::unique_lock<std::shared_mutex> lock(mutex_);
    value_++;
  }

  void reset() {
    std::unique_lock<std::shared_mutex> lock(mutex_);
    value_ = 0;
```

```
  }

 private:
  mutable std::shared_mutex mutex_;
  unsigned int value_ = 0;
};
```

### *4.2. Binários Literais*

Números são escritos normalmente na forma decimal. Agora, com a adição do prefixo "0b", o programador pode escrever números na forma binária.

```
int a = 0b00010;
```

### *4.3. Dedução de Tipagem*

Apesar de ser uma linguagem de tipagem forte, o *C++* tem suporte para definições dinâmicas de variáveis. Isso é feito através do tipo *auto*. Em outras palavras, se o programador declarar uma variável da seguinte forma, o compilador entende que a variável é do tipo *int*.

```
auto i = 0;
```

Isso já estava presente nas versões anteriores. Nesta, a dedução de tipagem foi melhorada para que também possa ser usada em funções.

```
auto add (int a, int b)
{
  return a + b;
}
```

Como ambos os parâmetros possuem tipo definido, o compilador entende que uma operação entre eles resultará em um valor do mesmo tipo. Neste caso, a função retornará um valor do tipo *int*.

Além disso, funções *lambda* também ganharam esta funcionalidade. Podemos declarar a seguinte função, muito parecida com a anterior, mas na forma de *lambda*.

```
auto lambda  = [](auto a, auto b) {return a + b};
```

Neste caso, o resultado terá o mesmo tipo de suas variáveis. Portanto, isso pode ser usado para qualquer tipo, como *int* ou *string*, e o resultado ainda será válido.

### *4.4. Templates em variáveis*

*Templates* agora podem ser utilizadas para declarar variáveis. No código abaixo, é possível definir o tipo retornado.

```cpp
template<typename T>
constexpr T a = T(1.5);

int func()
{
  a<int> = 1;
  std::cout << n<int> << " ";    // 1
  std::cout << n<float> << " ";  // 1.5
}
```

### *4.5. Sufixos*

Ao definir uma variável com *auto*, é possível adicionar um sufixo no fim do seu valor para que o compilador consiga saber seu tipo. Esses tipos são aqueles definidos na própria *stdlib*, e não os tipos naturais do *C++*. (ex: Ele lida com o tipo *std::basic string*, e não com *string*.)

Esta *feature* já estava presente no *C++ 11*, mas foi extendida e possui definições na biblioteca padrão agora. Antes, todos os sufixos deviam ser definidos pelo usuário.

```cpp
auto a = 1i; // numero complexo double
```

### *4.6. Separador de Dígitos*

Ao escrevermos números grandes no cotidiano, é normal utilizarmos separadores indicando a casa dos milhares, milhões, etc... (ex: 1.000). Para facilitar a leitura do código, agora números podem ser escritos com separadores também, sendo que eles não afetam de nenhuma forma a execução e a compilação do código.

```cpp
int mil = 1'000;
```

### *4.7. Deprecated*

Um atributo usado para indicar que uma entidade foi descontinuada. Isso serve como sinalizador para outros programadores verem que, apesar de uma certa função ou variável estar implementada, ela pode não ser a melhor solução. Além disso, uma mensagem pode ser adicionada (sem precisar de comentário) para explicar o motivo da descontinuação.

Este atributo não interfere na execução do programa. Em alguns compiladores, um *warning* será exibido ao compilar o programa, avisando sobre o uso de uma entidade descontinuada. Esta funcionalidade foi incluída baseada em atributos semelhantes de outras linguagens de programação, como *Java*.

```cpp
[[deprecated]] int func();
```

## 5. C++ 17

### *5.1. Retorno múltiplo*

Uma função agora pode retornar mais de um valor.

```
auto [a, b] = getValues();
```

### *5.2. Novos tipos*

Alguns tipos de variáveis foram adicionados à biblioteca padrão. São eles:

- string view: String normal, mas cujo valor não pode ser alterado.
- optional: Parâmetros opcionais.
- any: Valor de qualquer tipo.