Programação em C++

Kayky Moreira Praxedes

Fevereiro 2026

1 Características da linguagem

C++ é uma linguagem multiparadigma (suporta programação procedural, genérica e orientada a objetos) **de alto nível** (possui muitas abstrações, mas ainda permite gestão manual de hardware) extremamente rápida (devido à proximidade com a *CPU*), com muita presença em sistemas operacionais e embarcados, e programação de alto desempenho.

Para a execução do código, **o arquivo *.cpp* tem de ser compilado** (o compilador mais utilizado é o *g++*), gerando um arquivo executável.

A função principal do programa é o `main`, responsável por definir as ações realizadas pelo programa (função ativa) [01], as demais funções servindo como ferramentas (função passiva).

A biblioteca `iostream` é utilizada em muitas aplicações de C++, pois **possui funções para a utilização *input* e *output*** (com formatação automática), necessário para a grande maioria dos programas. Suas funções podem ser acessados pelo *namespace* **`‘‘std’’`**.

Formatação do código:

Espaçamentos e comentários não alteram diretamente funcionamento do código (apenas melhoram a legibilidade e organização do programa). **Instruções são delimitadas com um ponto e vírgula (;), e funções e estruturas de controle** (blocos como condicionais e *loops*) **são delimitados por chaves ({ })** [02].

```
#include <iostream>  // Biblioteca para o uso de elementos de input e output
/* Comentário com
mais de uma linha */
// Comentário com apenas uma linha
int main(){ // Função de execução do programa com retorno int e sem argumentos
    std::cout << "Hello, World!" << std::endl; /* Instrução do iostream que imprime uma mensagem no terminal,
    finalizada por um ponto e vírgula (;) (output: Hello, World!) */
    return 0; // Instrução de retorno da função (uso facultativo no main)
} // Domínio da função delimitado por chaves ({})
```

2 Tipos fundamentais de dados

Tratam-se dos tipos de dados padrão da linguagem, indicando as propriedades de uma variável, restringindo algumas operações e indicando o espaço de memória reservado [03] (variando a depender da arquitetura). Em C++, todos os dados são essencialmente numéricos (números binários), variando o especificador de tipo. Essa propriedade facilita operações de tipos diferentes, bem como sua conversão [04].

- **`char`:** Caracteres (dentro da tabela ASCII) [05]. Normalmente a memória reservada é de 1 *byte* (8 *bits*).
- **`int`:** Números inteiros. Normalmente a memória reservada é de 2 a 4 `bytes`.
- **`float`:** Números racionais. Normalmente a memória reservada é de 4 *bytes*.
- **`double`:** Também são números racionais, mas com o dobro de capacidade do `float` (nesse caso, 8 *bytes*).
- **`bool`:** Indicam valores de verdadeiro e falso [06]. A memória reservada é de 1 *byte*.

3 Armazenamento de informações

O armazenamento das informações voláteis é feito através de variáveis (em C++ são rótulos para endereços de espaços de memória associados a tipos [07]). Primeiro declara-se a variável (aloca seu espaço de memória), depois atribui-se um valor a ela (condizente com o tipo da variável) [08], podendo ser um valor literal, ou indireto (resultado de retorno de uma função, valor de outra variável, etc.).

```cpp
#include <iostream>

int main(){ // void implícito nos argumentos
    char a; // Declaração de uma variável (possui lixo de memória)
    a = {'c'}; // Atribuição de uma lista com um elemento literal (char) à variável
    double b{}; // Declaração de uma variável vazia (0.0)
    char c{a}; // Declaração com atribuição de uma lista com um elemento (variavel)
    int d = 20.5, e{40 + 20}; // Declaração múltipla com atribuição de valor (com casting) e expressão
    std::cout << "a = " << a << ", b = " << b << ", c = " << c << ", d = " << d << ", e = " << e << std::endl;
    // output: a = c, b = 0, c = c, d = 20, e = 60
    std::cin >> b >> a; // Instrução da iostream que recebe valores no terminal e os atribui às variáveis
    // Supondo o input: 20.5 d
    std::cout << "a = " << a << ", b = " << b << std::endl; // output: a = d, b = 20.5
} // O std::endl serve para indicar o fim da linha (quebra linha no terminal)
```

Constantes:

Elementos cujo valor após sua declaração não pode ser modificado, sendo esse o único momento onde é possível atribuir um valor à ela.

Elementos locais e globais:

Se um elemento (variável ou constante) **é declarado dentro de um bloco, trata-se de um elemento local** (a memória dessa variável só fica reservada apenas durante a execução do bloco, e essa é acessível diretamente apenas pelo bloco onde foi declarada e seus sub-blocos), **se não, trata-se de um elemento global** (seu endereço de memória fica alocado durante toda a execução do código e esse pode ser acessado por qualquer função).

Elementos pertencentes aos mesmos blocos e sub-blocos não podem ter o mesmo nome, mas podem ter elementos com nomes iguais em blocos diferentes. A prioridade de acesso é do dado local, sendo necessário colocar :: na frente do nome para acessar o elemento global.

```cpp
#include <iostream>

const int constanteGlobal{10};
int variavelGlobal1{20}, variavelGlobal2{};

int main(){
    const int constanteLocal{30};
    int variavelLocal{40};
    int variavelGlobal1{50}; // Variável local com o mesmo nome da global
    std::cout << "Local: " << variavelGlobal1 << ", Global: " << ::variavelGlobal1 << std::endl;
    // output: Local: 50, Global: 20
    ::variavelGlobal1 = {60}; // Atribuição para elemento global com nome igual a um elemento local
    variavelGlobal2 = {70}; // Atribuição para elemento global com nome único
    for(int i{0}; i < 10; i++){
        // Operação qualquer
    }
    int i{5}; // Como a variável i só existia no for, seu nome pode ser reaproveitado agora
}
```

4 Operações

As informações contidas nas variáveis podem ser manipuladas através de operações predefinidas na linguagem, essas seguindo uma ordem de prioridade que, a grosso modo, seguem a hierarquia: [09]

1. **Parênteses**.
2. **Operadores aritméticos, lógicos, comparativos, etc**.
3. **Operadores de atribuição**.

É bom definir explicitamente a ordem das operações por parênteses para evitar *bugs* e comportamentos inesperados.

5 Elementos de tomada de decisão

Elementos que realizam **diferentes ações a depender do valor de seu escopo**.

`if/else`:

Estrutura de escolha binária (apenas duas opções, verdadeiro ou falso). Uma condição é definida no `if` que, se verdadeira, realiza a ação do bloco, mas se for falsa, passa-se para o próximo teste (`else if` ou `else`).

```cpp
#include <iostream>

int main(){
    int valor;
    std::cin >> valor;
    if(valor == 10){ // Se valor == 10, a comparação retorna true, se não, false
        std::cout << "É 10!" << std::endl;
        return 10;
    } else if(valor == 20 || valor == 30) // Chega nesse teste se valor != 10
    return 30;
    else { // valor != 10 && valor != 20 && valor != 30
        if('b') // Como o valor no escopo ('b') é diferente de 0, será executado sempre
            std::cout << "Ação alcançada!" << std::endl;
        else return 1; // Condição nunca alcançada
    }
}
```

`switch case`:

Estrutura que executa diferentes ações a depender do valor no escopo. Todos os `cases` necessitam obrigatoriamente um `break` (menos o `default`), se não, todos os cases abaixo também são executados (*fall-through*).

```cpp
#include <iostream>

int main(){
    int valor;
    std::cin >> valor;
    switch (valor){
        case 10: // valor == 10
        // Ação 1
        break;
        case 30: case 40: // valor == 30 || valor == 40
        // Ação 3
        break;
        default: // valor != 10 && valor != 30 && valor != 40
        // Ação padrão
        break; // break facultativo
    }
}
```

6 Elementos de repetição (*loops*)

Realizam as ações de seu bloco até que uma condição de parada (ou um `break`) **seja alcançada**.

`while`:

Realiza uma ação enquanto a condição do seu escopo for verdadeira, nem entrando no bloco se ela já for falsa. Existe uma variação dessa instrução, **`do while`, que executa o bloco antes de entrar no *loop*, repetindo-a enquanto a condição for verdadeira** (pelo menos uma vez irá executar a ação).

```cpp
#include <iostream>

int main(){
    int valor;
    std::cin >> valor;
    while(valor != 10){ // Se valor == 10 nem entra
        std::cout << "Valor diferente de 10" << std::endl;
        std::cin >> valor;
    }
    do{ // Quando sai do while, valor == 10
        std::cout << "Valor: " << ++valor << std::endl; // output: Valor: 11
    } while (valor <= 10); // Realiza a ação pelo menos uma vez, mesmo se valor >= 20 antes do bloco
}
```

`for`:

Duração do *loop* delimitada no seu escopo. Nesse é definida uma variável e seu valor de início (podendo ser uma variável local do `for` ou uma já existente), sua condição de parada e uma operação, tudo separado por pontos e virgulas (;).

```cpp
#include <iostream>

int main(){
    int u{0};
    for (int i{0}; u < 20; i++){ // Incrementa variável interna do for (i) de 1 em 1
        u += 2;
        std::cout << u << " ";
    } std::cout << std::endl; // output: 2 4 6 8 10 12 14 16 18 20
    for(u; u >= 10; u -= 3) // Decrementa variável externa ao for (u) de 3 em 3
        std::cout << u << " ";
    std::cout << std::endl; // output: 20 17 14 11
}
```

Ferramentas para alterar o fluxo de *loops*:

O `break` sai do bloco de execução instantaneamente (válido para *loops* e `switch`). **O `continue` ignora o resto das instruções abaixo dele no *loop***, começando a próxima repetição (funciona apenas em *loops*).

7 Funções

Funções são conjuntos de instruções montadas pelo programador para realizarem uma ação ao serem chamadas no código. Elas têm de ser definidas acima das funções que as utilizam (o `main`, portanto, sendo a última), ter seu tipo de retorno definido (`void` caso não retorne nada) e podem admitir argumentos (variáveis locais que são declaradas no escopo que copiam os dados passados na chamada da função).

Protótipos:

Ferramenta que pré-compilam as funções, permitindo seu posicionamento livre no código. É semelhante ao processo de declarar uma variável e definir seu valor depois, sendo necessário declarar em seu escopo apenas o tipo das variáveis. O seu uso no programa é facultativo (escolha estilística ou organizacional), e os protótipos ainda devem ser declarados acima de funções que chamam sua função diretamente.

Funções `inline`:

O compilador realiza uma cópia do código da função, substituindo sua chamada pelo próprio código, permitindo uma execução mais rápida e direta, visto que, ao executar uma função, é realizada uma cópia temporária do código, que é alocado e desalocado na memória (mais demorado). Seu uso é especialmente interessante se uma função for ser muito utilizada ao longo do programa.

```cpp
#include <iostream>

double funcaoComPrototipo(int, double); // Protótipo
void funcaoSemPrototipo(){
    std::cout << "20 * 30.5 = " << funcaoComPrototipo(20, 30.5) << std::endl;
    return; // return vazio facultativo, já que o tipo de retorno é void
}
inline int somarSete(int i){ // Função que será copiada pelo compilador
    return i + 7;
}
int main(){
    funcaoSemPrototipo(); // output: 20 * 30.5 = 610
    int u{1};
    for(int i{0}; u < 50; i++){
        std::cout << u << " ";
        u = somarSete(u);
    }
    std::cout << std::endl; // output: 1 8 15 22 29 36 43
}
double funcaoComPrototipo(int inte, double rac){
    return static_cast<double>(inte) * rac; // retorno obrigatório
}
```

Referência:

Em C++ é possível criar um tipo de dado utilizado para referenciar outro elemento, sendo simplesmente uma nova forma de acessar aquela variável (ligação direta), de modo que a alteração em um é visto pelos dois [10].

```cpp
#include <iostream>

int main(){
    int a{10}, b{20}; // Variáveis qualquer
    int& c{a}; // Referência de int ligado à variável a
    c -= 5;
    std::cout << "a = "<< c <<", b = "<< b << ", c = "<< c << std::endl; // output: a = 5, b = 20, c(a) = 5
    c = b; // A referência recebe o valor de b, mas continua ligado à variável a
    c += 10;
    std::cout << "a = "<< c << ", b = "<< b << ", c = "<< c << std::endl; // output: a = 30, b = 20, c(a) = 30
}
```

Quando no escopo da função são declaradas variáveis, o valor passado na chamada é copiado para as variáveis locais (o dado original não sofre qualquer interferência), **sendo essa a passagem por valor**. Todavia, **se no escopo da função são declarados referências**, passa-se a lidar diretamente com as variáveis originais dentro da função (sem cópia), **esse sendo chamado de passagem por referência**.

```cpp
#include <iostream>

void funcaoValor(int valor){
    valor -= 5; std::cout << "valor = " << valor << std::endl;
}
void funcaoReferencia(int& valor){
    valor -= 5; std::cout << "valor = " << valor << std::endl;
}
int main(){
    funcaoValor(15); // output: valor = 10
    int a{10};
    funcaoValor(a); // Permitida a passagem de variáveis e valores literais (output: valor = 5 )
    std::cout << "a = " << a << std::endl; // output: a = 10
    funcaoReferencia(a); // Não seria permitido funcaoReferencia(15) (não é uma variável) (output: valor = 5)
```

```
    std::cout << "a = " << a << std::endl; // output: a = 5
}
```

Apesar da eficiência de passar e receber referências (evita a cópia desnecessária de dados), **não é recomendado para dados de tipos fundamentais** (economia porca) **e exige precauções para evitar *bugs*** (como declarar os parâmetros `const`, evitando alterações indevidas), sendo mais recomendada para elementos grandes.

Sobrecarga:

É possível criar funções com o mesmo nome, alterando seu funcionamento, tipo de retorno e argumentos [11]. Pode-se então criar várias funções com mesmo propósito para diferentes tipos de dados [12].

```
#include <iostream>

int volume(const int& x, const int& y, const int& z){ // Passagem por parâmetro
    return x * y * z;
}
double volume(const double x, const double y, const double z){ // Passagem por valor
    return x * y * z;
}
char volume(){
    return 'a';
}
int main(){
    std::cout << "O volume é: " << volume(10, 20, 30) << std::endl; // output: O volume é: 6000
    std::cout << "O volume é: " << volume(10.5, 30.5, 13.0) << std::endl; // output: O volume é: 4163.25
    std::cout << "O volume é: " << volume() << std::endl; // output: O volume é: a
}
```

Argumentos `default`:

Podem ser predefinidos valores padrão de argumentos para a função, esses sendo utilizados quando não há uma passagem explícita na chamada da função [13].

```
#include <iostream>
// utilização do const para evitar qualquer mudança indesejada na variável pelo parâmetro
int volume(const int& x = 1, const int& y = 1, const int& z = 1){
    return x * y * z;
}
int main(){
    std::cout << volume() << std::endl; // volume(1, 1, 1) implícito (output: 1)
    std::cout << volume(10) << std::endl; // volume(10, 1, 1) implícito (output: 10)
    std::cout << volume(10, 20) << std::endl; // volume(10, 20, 1) implícito (output: 200)
    std::cout << volume(10, 20, 30) << std::endl; // volume(10, 20, 30) (output: 6000)
}
```

Recursão:

Trata-se de um tipo especial de operação onde **uma função realiza uma chamada de si mesma, gerando algo semelhante a um *loop***. Quando a condição de parada é atingida, é encerrado o processo recursivo, permitindo que as chamadas retornem seus resultados gradualmente [14].

```
#include <iostream>

void fibonacciRecursivo(int antecessor, int atual, int termo){
    std::cout << antecessor << ", ";
    if(--termo > 1) // Chamada recursiva dos n primeiros termos da sequência
        fibonacciRecursivo(atual, antecessor + atual, termo);
    else std::cout << atual << std::endl; // Fim da chamada
}
int main(){
    fibonacciRecursivo(1,1,15); // Imprime 15 termos da sequencia de fibonacci
} // output: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610
```

Biblioteca padrão:

O C++ disponibiliza em sua biblioteca padrão um conjunto de ferramentas e funções prontas. Podem ser adicionadas por meio do comando `#include <nome da biblioteca>` (verificar apêndice B) e suas funções acessadas pelo *namespace* (elemento que indica a origem da função) `std` (origem na *standard library*) [15].

O uso de bibliotecas evita redundância de programação (não há necessidade de criar funções do zero, economizando tempo) **e *bugs*** (as funções já foram testadas e otimizadas, sendo mais eficientes e previsíveis), sendo muito encorajada principalmente com utilização de *POO*.

8 Introdução a programação orientada a objetos (*POO*)

Classes e objetos:

Classes permitem a definição de novos tipos de dados além dos tipos fundamentais. São compostas por atributos (dados) e métodos (funções), com diferentes níveis de acesso, como `public` (acesso externo) ou `private` (acesso restrito à própria classe) e normalmente são definidas em um *header* (também pode ser no código principal).

Para utilizar uma classe, é necessário criar objetos, que são instâncias desse tipo. Através deles, pode-se acessar os métodos e membros públicos da classe. Em geral, os atributos são declarados como `private`, sendo acessados indiretamente por métodos `public` (`getters` e `setters`), permitindo controle sobre o uso dos dados.

```
/* newClass.h */
class newClass{
    public: // Elementos que podem ser acessados pelo objeto instanciado
        void setConta(int conta){ // Método público para modificar a conta
            if(conta >= 0) minhaConta = conta; // Condição para a modificação da conta
        }
        void setSaldo(double saldo){ // Método público para modificar o saldo
            this->saldo = saldo; // o this especifica que trata-se do parâmetro da classe
        } // (semelhante ao :: para elementos globais em funções)
        int getConta() const { // Método público para retornar a conta
            return minhaConta; // const pois o método não deve alterar o saldo (prevenção)
        }
        double getSaldo() const { return saldo; } // Método público para retornar o saldo
    private: // Elementos que ficam inacessíveis diretamente ao objeto
        double saldo{}; // Elementos inicializados com valor igual à 0 por padrão
        int minhaConta{};
};

/* main */
#include <iostream>
#include <string> // Biblioteca com classe string
#include "newClass.h" // Biblioteca autoral

int main(){
    int a{10}; // Variável de tipo fundamental
    std::string objStr; // Objeto instanciado da classe string (vazio por padrão)
    objStr = {"Hello, World!"};
    std::cout << "objStr: " << objStr << std::endl; // output: objStr: Hello, World!
    newClass objNew; // Objeto instanciado da classe newClass
    std::cout << "Conta: "<< objNew.getConta() << ", saldo: "<< objNew.getSaldo() << std::endl;
    // output: Conta: 0, saldo: 0
    objNew.setConta(-5); objNew.setSaldo(1000); // Acesso das funções internas do objeto instanciado
    std::cout << "Conta: "<< objNew.getConta() << ", saldo: "<< objNew.getSaldo() << std::endl;
    // output: Conta: 0, saldo: 1000
    objNew.setConta(25); objNew.setSaldo(-500);
    std::cout << "Conta: "<< objNew.getConta() << ", saldo: "<< objNew.getSaldo() << std::endl;
    // output: Conta: 25, saldo: -500
}
```

Interface x Implementação:

É interessante modularizar os elementos de um programa, como as classes (esse processo torna o código mais organizado, eficiente e reutilizável), podendo ser separadas em:

- **Interface:** *header* que irá conter o protótipo dos métodos e os parâmetros da classe.
- **Implementação:** Código que irá conter a definição dos métodos e será compilado junto ao arquivo principal (ocultando detalhes de implementação).

```
/* Time.h */
#include <string>
#ifndef TIME_H // Include guard
#define TIME_H // Previne que o mesmo arquivo seja incluído múltiplas vezes no mesmo código fonte.
/* Supondo classes derivadas do time.h (alarme.h, relogio.h, etc.), cada um com  uma chamada de time.h.
Usando múltiplas classes sem o include guard, o compilador veria a definição da classe Time três vezes no
mesmo arquivo, causando erro.*/
class Time{
    public:
        void setTime(int, int, int);
        std::string getTime24h() const;
        std::string getTime12h() const;
    private:
        unsigned int horas{}, min{}, seg{};
};
#endif

/* Time.cpp */
#include <string>
#include <iomanip> // Biblioteca para a manipulação do input e output
#include <sstream> // Biblioteca para a manipulação do stream (conversões de/para string)
#include "Time.h"

void Time::setTime(int horas, int min, int seg){
    if((horas >= 0 && horas < 24) && (min >= 0 && min < 60) && (seg >= 0 && seg < 60)){
        this->horas = horas;
        this->min = min;
        this->seg = seg;
    }
}
std::string Time::getTime24h() const{
    std::ostringstream output;
    output << std::setfill('0') << std::setw(2) << horas << ":" << std::setw(2) << min << ":" << std::setw(2)
    ↪   << seg;
    return output.str();
}
std::string Time::getTime12h() const{
    std::ostringstream output;
    output << std::setfill('0') << std::setw(2) << ((horas==0 || horas==12) ? 12 : horas%12) << ":" <<
    ↪   std::setw(2)
    << min << ":" << std::setw(2) << seg << ((horas > 12) ? " PM" : " AM");
    return output.str();
}

/* main */
#include <iostream>
#include "Time.h"

int main(){
    Time t1, t2;
    t1.setTime(13,29,15);
    std::cout << t1.getTime24h() << std::endl; // output: 13:29:15
    std::cout << t1.getTime12h() << std::endl; // output: 01:29:15 PM
    t2.setTime(99,99,99); // Declaração com argumentos inválidos
    std::cout << t2.getTime24h() << std::endl; // output: 00:00:00
    std::cout << t2.getTime12h() << std::endl; // output: 12:00:00 AM
}
```

Construtores:

É possível definir parâmetros e ações a serem tomadas durante a declaração do objeto através da adição de um construtor ao código da classe [16], sendo possível **inclusive fazer sobrecarga de construtores e definir valores** `default`, (lembrando que pra sobrecarga os argumentos devem ser diferentes) [17].

```
/* Time.h */
#include <string>
#ifndef TIME_H // Include guard
#define TIME_H
class Time{
    public:
    explicit Time(int = 0, int = 0, int = 0); // Construtor com elementos default
    // Resto da interface igual...
};
#endif

/* Time.cpp */
#include <string>
#include <iomanip>
#include <sstream>
#include "Time.h"

Time::Time(int horas, int min, int seg){ // Implementação do construtor
    setTime(horas, min, seg);
}
    // Resto da implementação igual...

/* main */
#include <iostream>
#include "Time.h"

int main(){
    Time t1(10), t2{20, 25}, t3(21, 10, 9); // Chamadas utilizando a sobrecarga de construtores
    std::cout << t1.getTime24h() << ", " << t2.getTime12h() << ", " << t3.getTime24h() <<std::endl;
    // output: 10:00:00, 08:25:00 PM, 21:10:09
    t1.setTime(13,29,15); std::cout << t1.getTime24h() << std::endl; // output: 13:29:15
}
```

9 Conjuntos de elementos

Em C++ os conjuntos são *containers* gerenciados pelo compilador (não manualmente como em C). Os principais conjuntos são:

- `array`: **Classe cujos objetos são conjuntos de elementos de mesmo tipo (incluindo tipos fundamentais e objetos) com tamanho fixo**. Seus dados são guardados sequencialmente na memória com seu início no termo 0.

```
#include <iostream>
#include <array>

int main(){
    // Iniciação de um objeto array de um tipo com n elementos (arr[0] - arr[n-1])
    std::array<int, 10> arr1, arr2{}; // Arrays de 10 elementos com lixo de memória e zerado
    arr1 = {10, 20, 30}; // Atribuição de lista ao array (arr1 = {10, 20, 30, 0, 0, 0, 0, 0, 0, 0})
    arr1[3] = {20}; // Atribuição lista à um elemento (arr1 = {10, 0, 0, 20, 0, 0, 0, 0, 0, 0})
    arr1.at(5) = {30}; // Outro método de atribuição de um elemento (arr1 = {10, 0, 0, 20, 0, 30, 0, 0, 0, 0})
    std::array<int, 10> arr3{arr2}; // Atribuição entre arrays apenas do mesmo tamanho
    // Array de múltiplas dimensões
    std::array<std::array<int, 3>, 3> arr2d{5, 7, 9, 10};
    // arr2d[0] = {5, 7, 9}, arr2d[1] = {10, 0, 0}, arr2d[2] = {0, 0, 0}
    arr2d[2][2] = 20; // arr2d[0] = {5, 7, 9}, arr2d[1] = {10, 0, 0}, arr2d[2] = {0, 0, 20}
    for(int i{}; i < arr2d.size(); i++) // função que retorna o número de elementos do array
        for(int j{}; j < arr2d[i].size(); j++)
            arr2d[i][j] = 3 * i + j + 1; // arr2d[0] = {1, 2, 3}, arr2d[1] = {4, 5, 6}, arr2d[2] = {7, 8, 9}
}
```

- `vector`: **`vectors` são basicamente `arrays` com tamanho dinâmico** (pode ser alterado).

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

int main(){
    std::vector<int> vec1(10); // Vetor com 10 elementos (já zerados)
    vec1[9] = 10; // vec1.at(9) = 10; (vec1 = {0, 0, 0, 0, 0, 0, 0, 0, 0, 10})
    std::vector<int> vec2 = vec1; // Atribuição independe do tamanho
    std::vector<int> vec3(15);
    vec3 = vec1; // Alteração de tamanho do vetor (vec3 = {0, 0, 0, 0, 0, 0, 0, 0, 9, 10})
    vec3.push_back(30); // Adição ao final (vec3 = {0, 0, 0, 0, 0, 0, 0, 0, 9, 10, 30})
    vec3.pop_back(); // Elimina o último elemento (vec3 = {0, 0, 0, 0, 0, 0, 0, 0, 9, 10})
}
```

Existem mais tipos de conjuntos, como `maps`, `lists`, etc. cada um com sua função.

Range-based `for`:

Conjuntos como `arrays`, `vectors`, `maps`, `lists`, etc. podem ter seus elementos percorridos pelo `for` de maneira simplificada. É definido um elemento do mesmo tipo do conjunto, que copiará os dados para si até o último elemento.

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

int main(){
    std::vector<int> vec1{1, 2, 3, 4, 5, 6, 7, 8};
    std::vector<int> vec2{vec1};
    for(int i{}; i < vec1.size(); i++) // for padrão
    vec1[i] *= 2; // Alteração direta dos dados em vec1
    for(int i : vec2) // Range-based for (i copia o valor de vec2 da posição)
    i *= 2; // vec2 se mantém inalterado
    for(int i : vec1) std::cout << i << " ";
    std::cout << std::endl; // output: 2 4 6 8 10 12 14 16
    for(int i : vec2) std::cout << i << " ";
    std::cout << std::endl; // output: 1 2 3 4 5 6 7 8
}
```

10 Ponteiros

Ponteiros são elementos especiais que armazenam em si o endereço de outras variáveis (que pode ser acessado pelo prefixo &), podendo alterá-las, assim como referências [18]. Para criar um ponteiro, basta colocar um asterisco (∗) ao lado do tipo de dado durante a declaração, e para acessar a variável basta colocar um (∗) antes da variável durante a operação. **Podem ser criados ponteiros para quaisquer tipos** [19], **até ponteiros de um ponteiro**, através de um duplo asterisco (∗∗) e assim por diante, **ou ponteiros de objetos**.

```cpp
#include <iostream>

int main(){
    int a{10}, b{50};
    int* ptr1; // Declaração de um ponteiro de int com lixo de memória
    int* ptr2{}; // Ponteiro vazio (equivalente a int* ptr2{nullPtr};)
    ptr1 = &a; // ptr1 recebe o endereço da variável a
    *ptr1 *= 2; // Mesmo que a *= 2;
    std::cout << a << std::endl; // output: 20
    int* ptr3{&b}; // Declaração direta do endereço para o ponteiro
    *ptr3 -= 40; std::cout << b << std::endl; // output: 10
    ptr3 = &a; // Varios ponteiros apontando para a mesma variável
    int** ptr4{&ptr3}; // Ponteiro para ponteiro de int
    // ptr4 == &ptr3: *ptr4 == ptr3 == &a: **ptr4 == *ptr3 == a
    **ptr4 += 5; std::cout << a << std::endl; // output: 25
}
```

11 Encapsulamento e abstração

12 Alocação dinâmica

13 Programação genérica

14 *C-style*

Apêndice A

Características da linguagem

[01] - **Por padrão, o tipo adotado no `main` é `int`**, por conta da padronização de uso do valor de retorno 0 como indicativo que a execução ocorreu de maneira bem sucedida. Todavia, diferentemente do C, no `main` de um programa em C++, o `return 0` é implícito (seu uso sendo facultativo).

[02] - **Estruturas de controle podem ser usadas sem chaves, executando apenas a instrução imediatamente subsequente**. É recomendado sempre o uso de chaves, a fim de evitar comportamentos inesperados.

Tipos fundamentais de dados

[03] - **Por vezes o espaço de memória padrão para tipos numéricos não é suficiente, de modo que algumas operações levam a um *overflow* ou *underflow*** (resultado aritmético incorreto pois não havia memória o suficiente para armazenar o valor completo). Para sanar esse problema, **é possível aumentar o tamanho de memória para esses dados**, através da adição do prefixo `long` ao na declaração variáveis (`long int`, `long double`, etc.).

[04] - **Ao se atribuir um valor de um tipo à uma variável de outro tipo, ocorre um fenômeno chamado conversão implícita**, operada pelo próprio compilador. Essa pode levar a comportamentos indesejados no programa, visto que normalmente ela leva a uma perda de informações (ao atribuir 3.14 à uma variável `int`, o valor será arredondado para 3). **A conversão explícita dos dados é feita através do comando** `static_cast<novo tipo>(variável)`.

[05] - Como a tabela ASCII associa códigos numéricos aos caracteres, **operações aritméticas como adição e subtração podem ser realizadas com os elementos `char`** (especialmente útil para a formatação de caracteres e para cifragem).

[06] - **Outros tipos podem ser convertidos implicitamente para `bool`** (0 = `false`, diferente de 0 = `true`).

Armazenamento de informações

[07] - **Uma variável pode ser inicializada com o prefixo `auto`, que identifica o tipo de dado através da atribuição** (que tem de ocorrer na declaração). Todavia, seu uso não é recomendado pois pode acabar causando comportamentos inesperados no código devido à incompatibilidade de dados e conversões implícitas.

[08] - **Em C++ o método padrão de atribuição de dados é através listas** (elementos dentro de parênteses ({})), mesmo que para elementos unitários. Esse método previne o *casting* automático (conversão implícita de dados), sendo assim mais segura.

Enquanto nenhum valor for atribuído à variável após ela ser declarada, ela não estará vazia, mas sim conterá lixo de memória (dados soltos que se encontravam no endereço da variável). Desse modo, é necessário atribuir, mesmo que um valor nulo, um valor à variável antes de utilizá-la.

Atribuir chaves vazias (lista vazia) **é o equivalente à atribuir o elemento nulo** (maneira de representar ausência de valor) **à variável**, diferentemente representado em cada tipo (0 para elementos numéricos como `int` e `double`, `'\0'` para `char`, `nullPtr` para ponteiros, etc.).

Operações

[09] - Tabela completa da ordem de operações:

1. **Escopo e resolução de nomes:** operador de resolução de escopo (::).
2. **Parênteses:** ( ).
3. **Elementos que acessam valores:** acesso a arrays ([ ]), chamadas de função, acesso a membros de classe (.), acesso a membros via ponteiro ($->$), ponteiro para membro ($.*$ e $->*$).

4. **Operadores unários:** inversor de sinal ($-$), *NOT* lógico (!), *NOT bit* a *bit* ($\tilde{}$), incremento ($++$), decremento ($--$), operador de endereço (&), desreferenciação de ponteiros ($*$), `sizeof`, `alignof`, `typeid`, `new`, `delete`, `static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`. São avaliados da direita para a esquerda.
5. **Operadores aritméticos:** multiplicação ($*$), divisão ($/$) e resto da divisão (%) possuem prioridade maior que adição ($+$) e subtração ($-$).
6. **Deslocamento de bits:** esquerda ($<<$) e à direita ($>>$).
7. **Operadores relacionais:** maior ($>$), maior ou igual ($>=$), menor ($<$), menor ou igual ($<=$), igual ($==$), diferente ($!=$).
8. **Operadores bit a bit:** *AND* (&), *XOR* ($\hat{}$), *OR* (|).
9. **Operadores booleanos:** *AND* lógico (&&) e *OR* lógico (||).
10. **Operador condicional:** (? :). Funciona da seguinte maneira:
(condição) ? (valor se verdadeiro) : (valor se falso).
11. **Operadores de atribuição:** simples ($=$) e compostas ($+=$, $-=$, $*=$, $/=$, $\%=$, $\&=$, $|=$, $\hat{}=$, $<<=$, $>>=$). Esses operadores são avaliados da direita para a esquerda.
12. **Vírgula:** (,).

Os operadores unários de incremento e decremento, quando à esquerda da variável, realizam a operação e depois retornam o valor. Quando à direita, primeiro retornam o valor depois realizam a operação.

Funções

[10] - Uma referência tem de estar ligado a um valor desde a sua declaração, não podendo "ficar vazio", nem ser atribuído um valor literal a ele (no escopo de funções com o prefixo `const` é permitido).

[11] - O compilador diferencia as funções e escolhe qual usar pelos argumentos (seja pelo número de argumentos passados, pela sua ordenação ou pelos seus tipos).

```cpp
#include <iostream>

int funcaoQualquer(int a, char b, double c){
    // Operação da função
}
int funcaoQualquer(double a, int b, char c){
    // Operação da função
}
int funcaoQualquer(int a, int b){
    // Operação da função
}
int funcaoQualquer(double a, double b){
    // Operação da função
}
int main(){
    funcaoQualquer(10, 'a', 20.5); // Primeira versão
    funcaoQualquer(17.0, 10, 'c'); // Segunda versão
    funcaoQualquer(40, 55); // Terceira versão
    funcaoQualquer(27.7, 10.0); // Quarta versão
}
```

[12] - **É possível criar uma função que funciona independente do tipo** (evitando muitas sobrecargas da mesma função), **através de funções** `template`. Essa função normalmente é criada em um *header* (também pode ser feito no arquivo principal).

```cpp
/* maximo.h */
template <typename T> // or template<class T>
T maximo(T v1, T v2, T v3) { // Todos os elementos devem ser do mesmo tipo
    T max{v1};
    if(v2 > max) max = v2;
```

```
    if (v3 > max) max = v3;
    return max;
}

/* Código do main */
#include <iostream>
#include "maximo.h"
int main(){
    std::cout << "O máximo entre 10, 20 e -50 é: " << maximo(10, 20, -50) << std::endl;
    // output: O máximo entre 10, 20 e -50 é: 20
    std::cout << "O máximo entre 25.5, -30.2 e 41.1 é: " << maximo(25.5, -30.2, 41.1) << std::endl;
    // output: O máximo entre 25.5, -30.2 e 41.1 é: 41.1
    std::cout << "O máximo entre 'B', 'A' e 'C' é: " << maximo('B', 'A', 'C') << std::endl;
}    // output: O máximo entre 'B', 'A' e 'C' é: C`
```

[13] - É necessário tomar cuidado ao se trabalhar com sobrecarga principalmente ao se utilizar argumentos `default`, pois, implicitamente, é como se tivesse sido feita a sobrecarga considerando a falta de passagem de argumentos.

```
#include <iostream>

int volume(const int& x = 1, const int& y = 1, const int& z = 1){
    return x * y * z;
} /* Implicitamente, foi passado:
int volume(){int x{1}, y{1}, z{1};}
int volume(const int& x){int y{1}, z{1};}
int volume(const int& x, const int& y){int z{1};}
int volume(const int& x, const int& y, const int& z){} */
char volume(){ // duas versões da função volume com os mesmos parâmetros (volume())
    return 'a';
}
int main(){
    std::cout << "O volume é: " << volume() << std::endl; // Erro, pois o compilador não sabe qual escolher
}
```

[14] - **Apesar de chamadas recursivas conseguirem facilitar consideravelmente algumas operações em códigos, seu uso deve ser feito com cuidado**, visto que o processo deixa variáveis alocadas e ativas dentro de cada recursão. Para uma função com n variáveis, cada chamada recursiva adiciona um novo *frame* à *stack* (criando mais n variáveis).

O uso prolongado gera perda de eficiência no código. Se a memória for limitada, pode ocorrer um *stack-overflow* (memória completamente cheia).

[15] - **O *namespace* pode ser omitido da instrução**, deixando o código menos verboso, mas essa prática não é recomendada, pois pode causar conflitos de instrução (caso funções possuam o mesmo nome, mas de classes diferentes), e deixa o código menos organizado (visto que sem o *namespace*, não se sabe a origem da instrução).

```
#include <iostream>
using std::cout; // Permite que seja usado apenas o comando cout << mensagem...
using namespace std; /* Todas as funções com o namespace std são abreviadas:
std::cin >> variáveis... vira cin >> variáveis, std::endl vira endl, etc.*/

int main(){
    int a{};
    cin >> a; // Abreviação de todos os elementos com o namespace std
    cout << "O valor é " << a << endl; // Abreviação pontual para o std::cout
} // A instrução using std::cout; fica redundante já que using namespace std; abrevia ela e as demais
```

Introdução a programação orientada a objetos (*POO*)

[16] - Por padrão, **implicitamente a classe possui um construtor `default` vazio**, esse deixando de de ser acessível quando adicionado um ou mais construtores explícito.

[17] - O construtor pode ser utilizado implicitamente para passar valores para os parâmetros através de uma lista de dados. Para evitar esse comportamento, é necessário defini-lo como `explicit`.

```cpp
/* newClass.h */

class newClass{
    public:
    explicit newClass(int conta){ // Construtor usado apenas explicitamente durante a declaração do objeto
        if(conta >= 0) this->conta = {conta};
    }
    newClass(int conta, double saldo) : saldo{saldo}{ // Pode ser usado implicitamente
        if(conta >= 0) this->conta = {conta};
    }
    void setConta(int& conta){ if(conta >= 0) this->conta = conta; }
    void setSaldo(double& saldo){ this->saldo = saldo; }
    const int& getConta() const { return conta; }
    const double& getSaldo() const { return saldo; }
    private:
    double saldo{};
    int conta{};
};

/* main */
#include <iostream>
#include "newClass.h"

int main(){
    // Atribuições durante a declaração usando o construtor
    newClass obj1(10); // ou newClass obj1{10}, obj1.getConta() == 10, obj1.getSaldo() == 0
    newClass obj2(20, -300); // ou newClass obj2{20, -300}, obj2.getConta() == 20, obj2.getSaldo() == -300
    // Atribuição posterior à declaração usando o inplicitamente o construtor e uma lista de dados
    newClass obj3 = {30, 1000.0}; // obj3.getConta() == 30, obj3.getSaldo() == 1000
    obj2 = {50, 5000}; // obj2.getConta() == 50, obj2.getSaldo() == 5000
    /* Sem o explicit, seria possível, por exemplo: obj3 = {60}; obj3.getConta() == 60, obj3.getSaldo() == 0
    newClass obj4 = {70}, obj4.getConta() == 70, obj4.getSaldo() == 0*/
}
```

Ponteiros

[18] - **Apesar de ponteiros serem capazes de se comportar como referências, sendo possível inclusive utilizá-los para esse tipo de passagem**, o ponteiro ainda copia o endereço da variável (não é apenas um acesso direto por outra fonte) e sua existência não é condicionada a uma (podem se ligar a mais variáveis ou mesmo a nenhuma).

[19] - **A linguagem permite a criação de conjuntos sem uso de bibliotecas chamados** `built-in arrays` através do comando `tipo nome`$[n]$;. Também são sequenciais e de tamanho fixo, mas podem ser acessados por ponteiros (a variável é um ponteiro para o primeiro elemento), são mais limitados (não possuem funções internas, por exemplo), não sendo muito utilizado em C++ moderno (elemento *C-style*).

Apêndice B

Fontes

1. DEITEL, Harvey M.; DEITEL, Paul J. Como programar em C. 2. ed. Rio de Janeiro: LTC, 1994.
2. ZIVIANI, Nivio. Projeto de algoritmos com implementações em Pascal e C. 4. ed. São Paulo: Pioneira, 1999.
3. BATISTA, Natália Cosse. Ponteiros e alocação dinâmica de memória. 2022. 40 f. slides (PDF) da disciplina Algoritmos e estruturas de dados. Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG), 2025.
4. PEIXOTO, Daniela Cristina Cascini. Disciplina: Lógica de programação. Curso de graduação em Engenharia de Computação – CEFET-MG, 2024.
5. CAMPOS, Luciana Maria de Assis. Disciplina: Programação orientada a objetos. Curso de graduação em Engenharia de Computação – CEFET-MG, 2024.
6. BATISTA, Natália Cosse. Disciplina: Algoritmos e estruturas de dados. Curso de graduação em Engenharia de Computação – CEFET-MG, 2025.