

# IMD0030 – LINGUAGEM DE PROGRAMAÇÃO I

Aula 05 – Sobrecarga de Funções e Passagem de Parâmetro

---

# Objetivos da aula

- Introduzir os conceitos de sobrecarga de função e passagem de parâmetro por referência em **C++**
- Para isto estudaremos:
  - Funções com funcionalidades semelhantes e assinaturas distintas
  - Funções contendo argumentos padrão
  - Funções contendo referência à memória nos seus parâmetros
- Ao final da aula espera-se que o aluno seja capaz de:
  - Implementar diferentes tipos de funções utilizando sobrecarga e passagem por referência em **C++**

---

# Contexto

- Em linguagem C, cada função deve obrigatoriamente ter um nome único

```
int somarInt ( int x, int y) { return x + y; }  
float somarInt ( float x, float y) { return x + y; }  
double somarInt ( double x, double y) { return x + y; }
```

- Em linguagem C++, é possível repetir nomes de funções uma vez que o mecanismo de prototipagem permite identificar uma função por seu nome e seus argumentos

- **Mecanismo de sobrecarga**

```
int somar ( int x, int y) { return x + y; }  
float somar ( float x, float y) { return x + y; }  
double somar ( double x, double y) { return x + y; }
```

---

# Sobrecarga de funções (ou *overloading*)

- Sobrecarga de funções ocorre quando criamos duas ou mais funções com o mesmo nome, porém com assinaturas distintas
  - Polimorfismo Ad Hoc
  - Frequentemente utilizado na programação orientada a objetos

```
#include <iostream>
int somar ( int x, int y, int z) { return x + y + z; }
int somar ( int x, int y) { return x + y; }
float somar ( float x, float y) { return x + y; }
double somar ( double x, double y) { return x + y; }


int main () {
    std::cout << somar( 3, 2, 1) << std::endl;
    std::cout << somar( 1, 2) << std::endl;
    std::cout << somar( 7.5f, 8.0f) << std::endl;
    std::cout << somar( 2.0, 3.5) << std::endl;
    return 0;
}
```

---

# Sobrecarga de funções (ou *overloading*)

- A assinatura de uma função é composta pelo seu **nome**, **número** e **tipos de parâmetros**, sendo desconsiderado o tipo de retorno
  - Não podemos diferenciar funções apenas pelo tipo de retorno
  - Não podemos diferenciar funções apenas pelos nomes dos parâmetros

```
#include <iostream>
int somar ( int x, int y) { return x + y; } // OK
float somar ( int x, int y) { return x + y; } // Erro
double somar ( int a, int b) { return a + b; } // Erro
```



```
int main () {
    std::cout << somar( 2, 4) << std::endl;
    // Qual das funções o compilador deverá chamar?
    return 0;
}
```

---

# Sobrecarga de funções (ou *overloading*)

- Quando existirem duas ou mais funções com o mesmo nome, a decisão sobre qual delas será chamada é feita pelo compilador
  - Verifica se os tipos passados como argumentos casam com alguma das assinaturas da função
  - Ao verificar se há casamento, o compilador considera todas as possibilidades de conversões implícitas
- Exemplo:
  - Se na assinatura da função existe um parâmetro do tipo **int**
  - E um valor do tipo **char** é passado como argumento
  - O valor será aceito devido à conversão implícita de tipos

---

# Sobrecarga de funções (ou *overloading*)

- Supondo que os parâmetros sejam de tipos diferentes, a ordem deles também é relevante

```
#include <iostream>
float somar ( int x, float y) { return (float) x + y; }
float somar ( float x, int y) { return x + (float) y; }

int main () {
    std::cout << somar( 2, 4.1f) << std::endl;
    std::cout << somar( 2.1f, 4) << std::endl;
    return 0;
}
```

---

# Argumentos padrão

- Argumento padrão especifica um valor a ser utilizado quando o valor para um parâmetro for omitido na chamada de uma função
- Para definir um argumento padrão para um parâmetro, utiliza-se o símbolo **=** (igualdade) seguido de um valor na declaração do método

```
#include <iostream>
```

```
double corrigirPoupanca ( double valorInicial, double taxa = 0.5) {  
    return valorInicial * ( 1. + taxa / 100 );  
}
```

```
int main () {  
    // O segundo parâmetro recebe o valor padrão como argumento  
    double valorCorrigido = corrigirPoupanca( 1000.);  
    // O segundo parâmetro recebe um novo valor como argumento  
    double valorCorrigido = corrigirPoupanca( 1000., 0.6);  
    return 0;  
}
```



---

# Argumentos padrão

- Argumentos padrão devem se encontrar nos últimos parâmetros (em ordem) de uma função para que o compilador seja capaz de identificá-los
- Uma vez definido um argumento padrão para um parâmetro, todos os parâmetros seguintes também deverão ter um argumento padrão definido

```
#include <iostream>
```

```
double corrigirPoupanca ( double valorInicial, double taxa = 0.5) { ... } // OK
```

```
double corrigirPoupanca ( double valorInicial = 1000., double taxa = 0.5) { ... } // OK
```

```
double corrigirPoupanca ( double valorInicial = 1000., double taxa) { ... } // ERRO
```

// Obs: Como todas as funções acima têm a mesma assinatura, apenas uma delas poderia ser declarada em um programa

---

# Argumentos padrão

- **Cuidado:** Em certos casos, a identificação da função a ser chamada pode se tornar impossível para o compilador
  - Funções com assinaturas distintas que se tornem iguais com a omissão de alguns argumentos

```
double corrigirPoupanca ( double valorInicial ) { ... } // OK
```

```
double corrigirPoupanca ( double valorInicial, double taxa = 0.5) { ... } // ERRO – gera a mesma assinatura acima
```

```
int main() {  
    double valorCorrigido = corrigirPoupanca( 500. );  
    // Qual função o compilador deverá chamar?  
    // corrigirPoupanca( 500. ) ou corrigirPoupanca( 500. ) ?  
    return 0;  
}
```

---

# Revisão: passagem de parâmetros por valor

- Parâmetros de função são, por padrão (*default*), passados **por valor**
  - O valor (conteúdo) da variável utilizada na chamada é copiado para o conteúdo da variável local da função alocada no segmento de pilha
  - Como não existe cópia do endereço de memória, sendo somente o conteúdo da variável copiado, **não há como fazer alterações no conteúdo** da variável passada como parâmetro na chamada
  - A passagem por valor no **C** e no **C++** apresentam o mesmo comportamento

```
#include <iostream>
int somar ( int x, int y ) {
    return x + y;
}
int main() {
    int valor1 = 4, valor2 = 9;
    std::cout << somar( valor1, valor2 ) << std::endl;
    return 0;
}
```

# Revisão: passagem de parâmetros por nome

- A linguagem **C** não permite passagem de parâmetros por referência
  - Só é admitida passagem por valor de tipos escalares de dados
- Toda passagem por referência é **simulada através de ponteiros**
  - Simulada, pois na verdade, acaba sendo usada a passagem por valor (de ponteiros)
- Essa forma de passagem com o uso de ponteiros explícitos é chamada de **passagem por nome**
- No **C++** a passagem por nome segue os mesmos princípios do **C**
  - Mas o **C++** guarda uma nova carta na manga: **passagem por referência** (de verdade!!!)

```
1  #include <iostream>
2  void somar( int x, int y, int *resultado )
3  {
4      *resultado = x + y;
5  }
6
7  int main()
8  {
9      int valor1 = 4, valor2 = 9, resultado;
10     somar( valor1, valor2, &resultado );
11
12     std::cout << resultado << std::endl;
13     return 0;
14 }
15
```

---

# Passagem de parâmetros por referência

- Na passagem **por nome**, são explicitamente declarados ponteiros e passados endereços (que são manipulados explicitamente)
- Já passar **por referência** é o mesmo que declarar um argumento para ser um ponteiro e receber um endereço (de maneira implícita)
- Na passagem por referência, o **endereço** da variável utilizada na chamada é copiado para o conteúdo da variável local da função
  - Dentro da função, pode-se manipular os itens como se fossem objetos locais
  - Como existe cópia do endereço de memória, **alterações no conteúdo** da variável local são realizadas na mesma posição de memória da variável passada como parâmetro na chamada da função

# Passagem de parâmetros por referência

- A linguagem **C++** permite passagem de parâmetros por referência, tanto de tipos escalares quanto de objetos
- Para isto, o operador **&** precede o nome do parâmetro na declaração de um função
  - Com isso, o parâmetro passa a operar como um apelido para a variável passada na chamada da função

```
1  #include <iostream>
2  // simulação de passagem por referência em linguagem C
3  void somar( int x, int y, int *resultado ) { *resultado = x + y; }
4
5  // verdadeira passagem por referência em linguagem C++
6  void somar( int x, int y, int &resultado ) { resultado = x + y; }
7
8  int main()
9  {
10     int valor1 = 4, valor2 = 9, resultado;
11     somar( valor1, valor2, &resultado ); // chamada com uso de ponteiro
12     somar( valor1, valor2, resultado ); // chamada sem uso de ponteiro
13     std::cout << resultado << std::endl;
14     return 0;
15 }
16
```

# Passagem de parâmetros por referência

- A verdadeira passagem por referência elimina a necessidade do programador ter que **referenciar** (uso do operador **&**) e **desreferenciar** (uso do operador **\***) as variáveis a cada vez que precisar manipulá-las
  - Com referências, o compilador faz todo o trabalho, forçando o endereço a ser passado dentro de uma função
- Isto permite o desenvolvimento de código mais robusto (menos propenso a falhas) e de mais fácil leitura

```
1  #include <iostream>
2  void somar( int x, int y, int &resultado )
3  {
4      resultado = x + y;
5  }
6  int main()
7  {
8      int valor1 = 4, valor2 = 9, resultado;
9      somar( valor1, valor2, resultado );
10     std::cout << resultado << std::endl;
11     return 0;
12 }
```

---

# Resumo da aula

- Sobrecarga de funções é um conceito essencial quando precisamos propor funcionalidades semelhantes, mas com formas diferentes
- Passagem de parâmetros por referência permite alterar facilmente o conteúdo de qualquer variável em qualquer bloco de código
- A maneira como a passagem por referência é efetuada em linguagem **C++** é bastante natural, diferentemente da linguagem **C**
- Ambos são recursos poderosos que permitem flexibilidade no processo de desenvolvimento de software
  - Melhoria da reusabilidade do código desenvolvido
  - Melhoria substancial da portabilidade e legibilidade do código
  - Maior robustez
  - Menor custo e maior facilidade de manutenção



# Alguma Questão?



---

# Exercício de Aprendizagem

