

Programação de Computadores

TEMPLATES E SOBRECARGA DE FUNÇÕES

Introdução

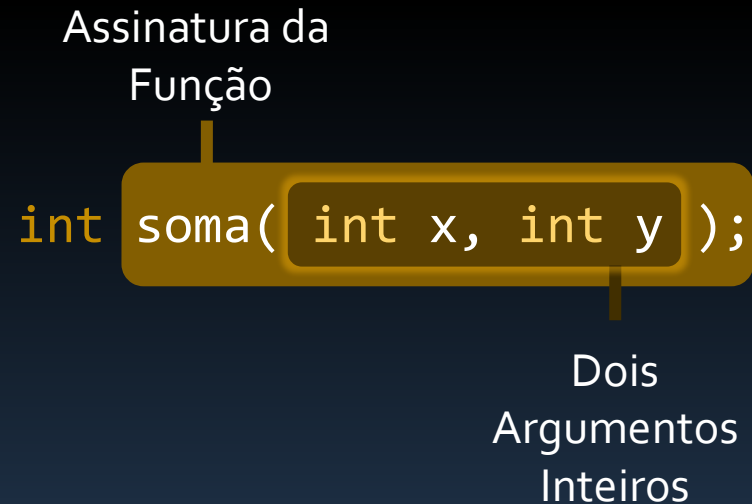
- Ao projetar uma função, o programador deve escrever um protótipo que estabelece claramente a **assinatura da função**:

- Nome da função
- Argumentos da função
 - Número
 - Tipo

Assinatura da
Função

```
int soma( int x, int y );
```

Dois
Argumentos
Inteiros

The diagram shows the function signature 'int soma(int x, int y);'. A yellow box highlights the entire signature. A line points from the text 'Assinatura da Função' to the box. Another line points from the text 'Dois Argumentos Inteiros' to the parameters 'int x, int y'.

Introdução

- Ao fixar a quantidade e tipo dos argumentos na função, **perde-se flexibilidade**
 - Para somar floats é preciso escrever outra função

```
int somaI(int x, int y);  
float somaF(float x, float y);
```

- C++ resolve esse problema de três formas[†]
 - Argumentos padrão
 - Sobrecarga de funções
 - Templates

Argumentos Padrão

- Um argumento padrão é um valor que é usado automaticamente no caso de **omissão do argumento real**

```
// protótipo da função linha  
void linha(char ch, int tam = 10);
```

- A função pode ser chamada com ou sem valores para o último argumento:

```
linha('-', 30); // linha tamanho 30  
linha('=');    // linha tamanho 10
```

Argumentos Padrão

- Ao usar uma lista de argumentos, os argumentos padrões devem vir **no final da lista**

```
void rubens(int i = 1, int j = 4, int k = 5);  ✓ // válido
void nelson(int i, int j = 4, int k = 5);      ✓ // válido
void ayrton(int i, int j = 6, int k);          ✗ // inválido
```

- Os argumentos são passados na ordem dos parâmetros
- Não tem como indicar que um parâmetro deve ser saltado

```
ayrton(5,2); // inicializa i e j
              // k não possui valor
```

Argumentos Padrão

- Ao usar uma lista de argumentos, os argumentos padrões devem vir **no final da lista**

```
void rubens(int i = 1, int j = 4, int k = 5);  ✓ // válido
void nelson(int i, int j = 4, int k = 5);      ✓ // válido
void ayrton(int i, int j = 6, int k);          ✗ // inválido
```

- A função nelson permite chamadas com 1, 2 ou 3 argumentos

```
nelson(2);           // o mesmo que nelson(2,4,5);
nelson(1,8);         // o mesmo que nelson(1,8,5);
nelson(8,7,6);       // não usa argumentos padrão
```

Argumentos Padrão

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

void linha(int tam = 10, char ch = '-');

int main()
{
    linha();
    linha(20);
    linha(30, '*');
    linha('*');          // utilização errada
}

void linha(int tam, char ch)
{
    for (int i=0; i < tam; ++i)
        cout << ch;
    cout << endl;
}
```

Argumentos Padrão

- A saída do programa:

```
-----  
-----  
*****  
-----
```

- O caractere **asterisco foi convertido para um número**
 - Não é permitido omitir argumentos que estejam a esquerda de um argumento fornecido

```
linha('*');           // utilização errada
```


Sobrecarga de Funções

- A sobrecarga permite escrever várias funções que **compartilham o mesmo nome**
 - Exige que as funções tenham **assinaturas diferentes**
 - Número de parâmetros diferentes
 - Tipos dos parâmetros diferentes

```
void exibir(const char * str, int tam);  
void exibir(float num, int tam);  
void exibir(long num, int tam);  
void exibir(int num, int tam);  
void exibir(const char * str);
```

Sobrecarga de Funções

- Ao usar uma função, o **compilador casa o tipo dos argumentos** com o tipo dos parâmetros

```
void exibir(const char * str, int tam);    // #1
void exibir(double num, int tam);         // #2
void exibir(long num, int tam);           // #3
void exibir(int num, int tam);            // #4
void exibir(const char * str);            // #5
```

```
exibir("panqueca", 15);                  // usa função #1
exibir("torta");                         // usa função #5
exibir(2420.0, 10);                      // usa função #2
exibir(2420, 12);                        // usa função #4
exibir(2420L, 15);                      // usa função #3
```

Sobrecarga de Funções

- É preciso usar o tipo de argumento correto

```
// chamada ambígua da função exibir  
unsigned int ano = 2014;  
exibir(ano, 6);
```

- O compilador não sabe para qual tipo deve converter **unsigned int**

```
void exibir(double num, int tam);           // #2  
void exibir(long num, int tam);             // #3  
void exibir(int num, int tam);              // #4
```

- Não haveria problema se existisse apenas uma opção

Sobrecarga de Funções

- Referências podem causar problemas

```
double cubo(double x);    // função recebe um double
double cubo(double & x);  // função recebe um double
```

- O compilador não é capaz de escolher qual função usar para a chamada abaixo:

```
double num = 2.0;
cout << cubo(num);
```

- Apesar da seguinte chamada funcionar:

```
cout << cubo(2.0);    // referência não-constante não pode receber 2.0
```

Sobrecarga de Funções

- O casamento discrimina **constantes de não-constantes**

```
void exibir(const char * str);    // com sobrecarga
void exibir(char * str);         // com sobrecarga
```

```
void mostrar(const char * str);  // sem sobrecarga
void apresentar(char * str);     // sem sobrecarga
```

```
const char con[] = "string constante";
char var[] = "string variável";
```

```
exibir(con);           // exibir(const char * str);
exibir(var);           // exibir(char * str);
mostrar(con);          // mostrar(const char * str);
mostrar(var);          // mostrar(const char * str);
apresentar(con); x    // nenhum casamento possível
apresentar(var);      // apresentar(char * str);
```

Sobrecarga de Funções

```
#include <iostream>
#include <cstring>
using namespace std;

unsigned tamanho(const char * str);
unsigned tamanho(unsigned num);

int main()
{
    const char * viagem = "Hawaii"; // string de teste
    unsigned n = 12345678;          // valor de teste

    cout << "Tamanho da string: ";
    cout << tamanho(viagem);
    cout << endl;

    cout << "Tamanho do número: ";
    cout << tamanho(n);
    cout << endl;
}
```

Sobrecarga de Funções

```
unsigned tamanho(const char * str)
{
    unsigned i = 0;
    while (str[i])
        i++;

    return i;
}
```

```
unsigned tamanho(unsigned num)
{
    unsigned digitos = 1;
    while (num /= 10)
        digitos++;

    return digitos;
}
```

Sobrecarga de Funções

- A saída do programa:

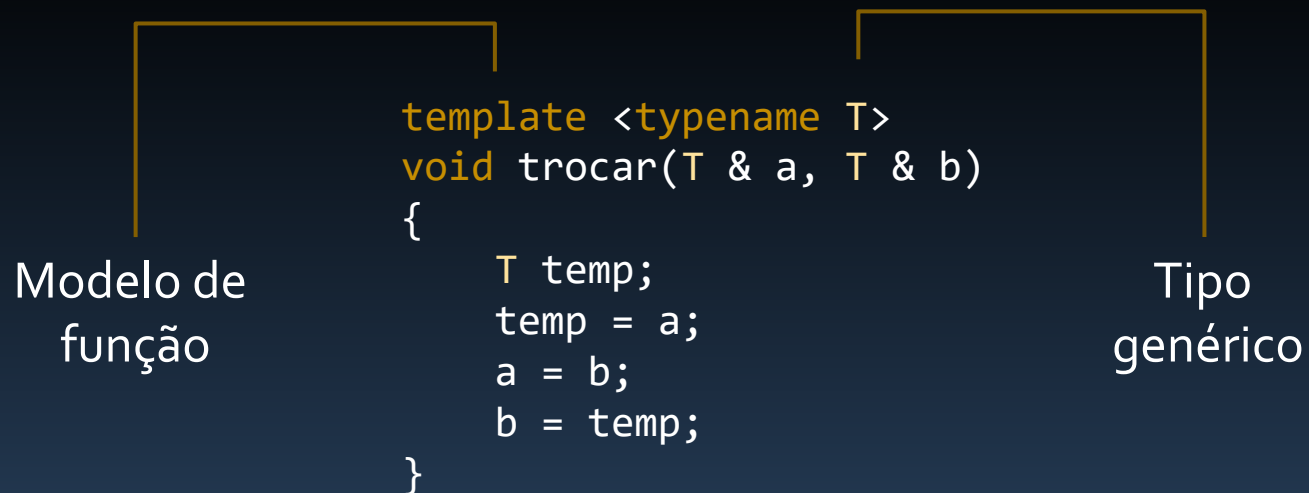
Tamanho da string: 6

Tamanho do número: 8

- Use sobrecarga para funções que fazem **tarefas semelhantes** com **tipos de dados diferentes**
- Prefira usar **argumentos padrão** à sobrecarga
 - Uso menor de memória para o programa
 - Apenas uma função para fazer manutenção

Templates de Funções

- Um **template** é uma **descrição genérica** de uma função
 - Ele é definido em termos de tipos genéricos, que podem ser substituídos por tipos reais
 - É a base para a **programação genérica**



Templates de Funções

- O template não cria nenhuma função, apenas **fornece um modelo para o compilador**
 - O compilador cria as funções com os tipos reais a partir das chamadas de funções do programa

```
template <typename T>
void trocar(T & a, T & b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

trocar(int,int);

```
void trocar(int & a, int & b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

trocar(double,double);

```
void trocar(double & a, double & b)
{
    double temp;
    temp = a;
    a = b;
    b = temp;
}
```

Templates de Funções

```
#include <iostream>
using std::cout;
using std::endl;

template <typename T> void trocar(T & a, T & b);

int main()
{
    int i = 10;
    int j = 20;
    double x = 24.5;
    double y = 81.7;

    cout << "valores originais:" << endl;
    cout << "i = " << i << ", j = " << j << endl;
    cout << "x = " << x << ", y = " << y << endl << endl;

    trocar(i, j);
    trocar(x, y);
}
```

Templates de Funções

...

```
cout << "valores trocados:" << endl;  
cout << "i = " << i << ", j = " << j << endl;  
cout << "x = " << x << ", y = " << y << endl;  
cout << endl;
```

```
return 0;
```

```
}
```

```
template <typename T>  
void trocar(T & a, T & b)  
{  
    T temp;  
    temp = a;  
    a = b;  
    b = temp;  
}
```

Templates de Funções

- A saída do programa:

valores originais:

i = 10, j = 20

x = 24.5, y = 81.7

valores trocados:

i = 20, j = 10

x = 81.7, y = 24.5

- O compilador criou duas funções:

```
void trocar(int & a, int & b);           // trocar(i,j);
```

```
void trocar(double & a, double & b);    // trocar(x,y);
```

Templates de Funções

- Use funções template quando precisar aplicar o **mesmo algoritmo** para argumentos de **tipos diferentes**

```
template <typename T>  
void trocar(T & a, T & b);
```

- É **importante** saber:
 - Templates não geram executáveis menores
 - O código gerado não contém templates

Sobrecarga de Templates

- Assim como é possível sobrecarregar funções, podemos **sobrecarregar templates**
 - O recurso é interessante para tratar tipos de argumentos com necessidades especiais

```
template <typename T>
void trocar(T va[], T vb[], int n)
{
    T temp;
    for (int i = 0; i < n; ++i)
    {
        temp = va[i];
        va[i] = vb[i];
        vb[i] = temp;
    }
}
```

Resumo

- C++ fornece maior flexibilidade com funções por meio de:
 - **Argumentos padrão**: alguns argumentos possuem valores predeterminados e podem ser omitidos na chamada da função
 - **Sobrecarga de função**: permite criar funções com o mesmo nome mas que agem em dados de tipos diferentes - **usada quando o tratamento é diferente para cada tipo**
 - **Template de função**: permite criar automaticamente funções com o mesmo nome que agem em dados de tipos diferentes - **usado quando o algoritmo é o mesmo para todos os tipos**