

Programação de Computadores

# FUNÇÕES COM STRINGS E REGISTROS

# Strings

- Strings são **sequências de caracteres**
  - São armazenadas em **vetores de caracteres**
  - O último caractere de toda string é o **caractere nulo** (escrito '**\0**', ele é o caractere de código ASCII 0)

```
char cachorro[4] = {'l', 'a', 't', 'e'};    // não é string
char gato[4]     = {'m', 'i', 'a', '\0'};  // string
```

0	1	2	3
l	a	t	e

✗

cachorro

0	1	2	3
m	i	a	\0

✓

gato

# Strings

- A **inicialização** de uma string pode ser simplificada usando uma **constante string**

```
char passaro[10] = "Gaivota"; // caractere \0 está implícito
char peixe[]      = "Sardinha"; // deixa o compilador contar
```

- Constantes string **entre aspas duplas** sempre incluem o **\0** implicitamente

```
char circo[8] = "Bozo";
```

0	1	2	3	4	5	6	7
B	o	z	o	\0	\0	\0	\0

caracteres '**\0**' são adicionados automaticamente

# Strings

- A string é **manipulada** pelo endereço do seu primeiro caractere

	0x38FD09
m	0x38FD0A = gato
i	0x38FD0B
a	0x38FD0C
\0	0x38FD0D
	0x38FD0E

```
char gato[4] = "mia";  
cout << gato; // mia
```

0	1	2	3
m	i	a	\0

gato

# Funções e Strings

- Uma string é armazenada em um vetor
  - O estudo de funções com vetores se aplica às strings
  - O parâmetro da função deve receber o endereço do início da string

```
// protótipo da função strlen?  
int strlen(char *);
```

- Pode-se usar const para proteger um argumento string contra modificações dentro da função

```
// protótipo de strlen melhorado  
int strlen(const char *);
```

# Funções e Strings

- Existem 3 possibilidades para **argumentos** tipo string:

- Um vetor de char

```
char vet[15] = "galopante";
```

- Uma constante string entre aspas duplas (*string literal*)

```
"galanteador"
```

- Um ponteiro para char apontando para uma string

```
char * str = "galáctico";
```

- Todas as opções são do tipo **ponteiro para char** (**char \***)

# Funções e Strings

```
char vet[15] = "galopante";  
char * str   = "galáctico";
```

```
int tam1 = strlen(vet);           // vet é o &vet[0]  
int tam2 = strlen(str);          // ponteiro para char  
int tam3 = strlen("galanteador"); // endereço da string
```

- Ao contrário dos vetores, não é necessário passar o **tamanho das strings** por parâmetro

```
int strlen(const char *);        // protótipo da função strlen  
int strlen(const char[]);       // protótipo da função strlen
```

# Funções e Strings

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

int CharEmString(char ch, const char * str);

int main()
{
    char espantado[15] = "uau!";           // string em vetor
    const char * admirado = "ulalalala!";  // admirado aponta para string

    int nu = CharEmString('u', espantado);
    int na = CharEmString('a', admirado);

    cout << nu << " caracteres u em " << espantado << "\n";
    cout << na << " caracteres a em " << admirado << "\n";

    return 0;
}
```



# Funções e Strings

```
int CharEmString(char ch, const char * str)
{
    int cont = 0;

    while (*str)    // encerra quando *str é '\0'
    {
        if (*str == ch)
            cont++;
        str++;      // move ponteiro para o próximo char
    }
    return cont;
}
```

- Saída do programa:

2 caracteres u em uau!

4 caracteres a em ulalalala!

# Funções e Strings

- A função `CharEmString` pode usar a **notação de vetor** tanto no parâmetro quanto dentro da função:

```
int CharEmString(char ch, const char str[])
{
    int cont = 0;

    for (int i=0; str[i]; i++)    // encerra quando str[i] é '\0'
    {
        if (str[i] == ch)
            cont++;
    }

    return cont;
}
```

# Retorno de Strings

- Funções não retornam strings

- Elas podem **retornar o endereço de strings**

```
// cuidado, retorno perigoso  
char * InverteString(const char * str);
```

- Uma função **nunca deve retornar** o endereço de **variáveis** ou **constantes string** criadas dentro da própria função

- A memória para constantes e variáveis locais é liberada ao final da execução da função

# Retorno de Strings

```
// método errado de retornar uma string
#include <iostream>
using namespace std;

char * InverteString(const char * str);

int main()
{
    char nome[40];
    cout << "Digite seu nome: ";
    cin >> nome;

    cout << "Seu nome invertido: ";
    cout << inverteString(nome) << endl;

    return 0;
}
```

# Retorno de Strings

```
char * inverteString(const char * str)
{
    char invertida[40];
    const int Tam = strlen(str);

    for (int i = 0; i < Tam; i++)
        invertida[i] = str[Tam-1-i];

    invertida[Tam] = '\0';

    return invertida;
}
```



- Saída do programa:

Digite seu nome: joaozinho

Seu nome invertido: r\0

# Retorno de Strings

- Existem **duas formas de retornar uma string** corretamente:

- Retornando o endereço de uma string alocada com **new**

```
char invertida[40];           // alocação estática  
char * invertida = new char[40]; // alocação dinâmica
```

- Passando um parâmetro adicional para ser modificado

```
void inverteString(const char * str, char * invertida)
```

- Vamos ver exemplos das duas soluções

# Retorno de Strings

- Usando **alocação dinâmica**

```
int main()
{
    char nome[40];
    cout << "Digite seu nome: ";
    cin >> nome;

    char * inv = InverteString(nome);
    cout << "Seu nome invertido: ";
    cout << inv << endl;
    delete [] inv;

    return 0;
}
```

```
char * InverteString(const char * str)
{
    const int Tam = strlen(str);
    char * invertida = new char[Tam+1];

    for (int i = 0; i < Tam; i++)
        invertida[i] = str[Tam-1-i];

    invertida[Tam] = '\\0';

    return invertida;
}
```

# Retorno de Strings

- Alocar memória dentro de uma função para ser liberada em outra função **não é uma boa ideia**
  - É fácil **esquecer o delete** e gerar um vazamento de memória

```
int main()
{
    ...
    char * inv = InverteString(nome);
    cout << "Seu nome invertido: ";
    cout << inv << endl;

    delete [] inv;
}
```

```
char * InverteString(const char * str)
{
    const int Tam = strlen(str);
    char * invertida = new char[Tam+1];

    ...

    return invertida;
}
```



# Retorno de Strings

- Usando um **parâmetro adicional**

```
int main()
{
    char nome[40], invertida[40];
    cout << "Digite seu nome: ";
    cin >> nome;

    InverteString(nome, invertida);
    cout << "Seu nome invertido: ";
    cout << invertida << endl;

    return 0;
}
```

```
void InverteString(const char * str,
                  char * inv)
{
    const int Tam = strlen(str);

    for (int i = 0; i < Tam; ++i)
        inv[i] = str[Tam-1-i];

    inv[Tam] = '\0';
}
```

# Registros

- **Registros** são ideais para **guardar**:
  - Informações de tipos diferentes
  - Agrupadas sob um único nome

**Ex.:** armazenar informações  
sobre um **jogador**

Nome  
Salário  
Altura  
Peso  
Gols

# Registros

- Declaração de um registro:

Palavra chave struct

Nome do registro

```
struct jogador  
{  
    char nome[40];  
    float salario;  
    unsigned gols;  
};
```

Membros do Registro

Finaliza a instrução de declaração

# Funções e Registros

- Quando se tratam de funções, os registros se comportam como os tipos básicos da linguagem C++

```
jogador bebeto = {"Bebeto", 600000, 800};
```

- Podem ser passados como argumentos

```
void mostrarJogador(jogador j);
```

- Podem ser retornados

```
jogador lerJogador();
```

# Funções e Registros

- Os registros são **passados por valor**

- A função recebe uma cópia do registro

```
cout << somaGols(bebeto, romario) << endl;
```

```
int somaGols(jogador j1, jogador j2);
```

- Uma alternativa é passar o endereço do registro

- A função deve **usar ponteiros** nos parâmetros

```
cout << somaGols(&bebeto, &romario) << endl;
```

```
int somaGols(jogador * j1, jogador * j2);
```

# Funções e Registros

- Passar um registro **por valor** só faz sentido quando ele é relativamente **pequeno**

```
struct tempo
{
    int horas;
    int mins;
};
```

- Considere o problema de calcular o tempo de uma viagem:

```
tempo SomaTempo(tempo t1, tempo t2);
void MostraTempo(tempo t);
```

# Funções e Registros

```
// usando registros com funções
#include <iostream>
using namespace std;

struct tempo
{
    int horas;
    int mins;
};

const int MinsPorHora = 60;

tempo SomaTempo(tempo t1, tempo t2);
void MostraTempo(tempo t);
```

```
int main()
{
    tempo dia1 = {5, 45};
    tempo dia2 = {4, 55};

    tempo viagem = SomaTempo(dia1, dia2);
    cout << "Total de dois dias: ";
    MostraTempo(viagem);

    tempo dia3 = {4, 32};

    cout << "Total de três dias: ";
    MostraTempo(SomaTempo(viagem, dia3));
}
```

# Funções e Registros

```
tempo SomaTempo (tempo t1, tempo t2)
{
    tempo total;
    total.mins = (t1.mins + t2.mins) % MinsPorHora;
    total.horas = t1.horas + t2.horas + (t1.mins + t2.mins) / MinsPorHora;
    return total;
}

void MostraTempo (tempo t)
{
    cout << t.horas << " horas, " << t.mins << " minutos\n";
}
```

- Saída do programa:

Total de dois dias: 10 horas, 40 minutos

Total de três dias: 15 horas, 12 minutos



# Funções e Registros

- Quando o registro guardar uma grande quantidade de informações, o ideal é **passar para a função apenas o endereço do registro**
  - Obtém-se o endereço de um registro usando o **operador &**

```
tempo SomaTempo(tempo * t1, tempo * t2);
```

```
tempo a = {3, 40};  
tempo b = {2, 10};  
tempo c = SomaTempo(&a, &b);  
MostraTempo(&c);
```

# Funções e Registros

- Se a **função continua retornando um registro**, continua-se com uma cópia de uma grande quantidade de dados
  - A solução é passar um terceiro argumento para ser modificado

```
void SomaTempo(tempo * t1, tempo * t2, tempo * soma);
```

```
tempo a = {3, 40};  
tempo b = {2, 10};  
tempo c;  
SomaTempo(&a, &b, &c);  
MostraTempo(&c);
```

# Funções e Registros

```
// Usando registros com funções
#include <iostream>
using namespace std;

struct tempo
{
    int horas;
    int mins;
};

const int MinsPorHora = 60;

void SomaTempo(const tempo * t1,
               const tempo * t2,
               tempo * t3);

void MostraTempo(const tempo * t);
```

```
int main()
{
    tempo dia1 = {5, 45};
    tempo dia2 = {4, 55};
    tempo dois;

    SomaTempo(&dia1, &dia2, &dois);
    cout << "Total de dois dias: ";
    MostraTempo(&dois);

    tempo dia3 = {4, 32};
    tempo tres;
    SomaTempo(&dois, &dia3, &tres);

    cout << "Total de três dias: ";
    MostraTempo(&tres);
}
```

# Funções e Registros

```
void SomaTempo (const tempo * t1, const tempo * t2, tempo * ret)
{
    ret->mins = (t1->mins + t2->mins) % MinsPorHora;
    ret->horas = t1->horas + t2->horas + (t1->mins + t2->mins) / MinsPorHora;
}
```

```
void MostraTempo (const tempo * t)
{
    cout << t->horas << " horas, " << t->mins << " minutos\n";
}
```

- Saída do programa:

Total de dois dias: 10 horas, 40 minutos

Total de três dias: 15 horas, 12 minutos

# Funções e Objetos `string`

- Objetos da **classe `string`** se parecem bastante com registros
  - Podem ser usados como um tipo básico da linguagem
    - Uma `string` é **passada por cópia**
    - Uma `string` **pode ser retorno** de uma função

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

string inverta(string s);
void inverta(const string * fonte, string * destino);
```

# Funções e Objetos string

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

void mostrar(const string vet[], int n);

int main()
{
    string planetas[5];
    cout << "Digite seus 5 planetas favoritos:\n";
    for (int i=0; i < 5; i++)
    {
        cout << i + 1 << ": ";
        getline(cin, planetas[i]);
    }

    cout << "\nSua lista:\n";
    mostrar(planetas, 5);
}
```

# Funções e Objetos string

```
void mostrar(const string vet[], int n)
{
    for (int i=0; i < n; i++)
        cout << vet[i] << " ";
}
```

## ■ Saída do programa:

Digite seus 5 planetas favoritos:

1: Terra  
2: Júpiter  
3: Marte  
4: Venus  
5: Saturno

Sua lista:

Terra Júpiter Marte Venus Saturno

# Resumo

- Ao passar **strings e vetores** para funções:
  - Manipula-se a **cadeia original** e não uma cópia
  - Para proteger os dados contra alterações pode-se usar **const**

```
void InverteString(const char * str, char * inv)
```

- Os **registros e objetos** são passados por cópia
  - Para evitar a cópia de um grande volume de dados é preciso **passar o endereço (&)** ou usar **referências**<sup>†</sup>