

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

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

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

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

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

Funções e Strings

- Existem 3 possibilidades para **argumentos** tipo string:
 - Um vetor de char

```
char vet[15] = "galopante";
```
 - Uma constante string (*string literal*)

```
"galanteador"
```
 - Um ponteiro para char

```
const char * str = "galáctico";
```
- Todas as opções são do tipo **char***

Funções e Strings

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

```
unsigned strlen(const char *);    // protótipo da função strlen  
unsigned 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;
}
```

CharEmString.cpp

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 * Inverter(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 * Inverter(const char * str);

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

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

    return 0;
}
```

InverteStr.cpp

Retorno de Strings

```
char * Inverter(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: **h o i n z o a o j**

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 Inverter(const char * str, char * invertida)
```

- Vamos ver prós e contras de cada solução

Retorno de Strings

- Usando **alocação dinâmica**

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

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

    return 0;
}
```

```
char * Inverter(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 = Inverter(nome);
    cout << "Seu nome invertido: ";
    cout << inv << endl;

    delete [] inv;
}
```

```
char * Inverter(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;

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

    return 0;
}
```

```
void Inverter(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

```
struct Jogador
{
    char nome[40];
    float salario;
    unsigned altura;
    float peso;
    unsigned 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 Exibir(Jogador j);
```

- Podem ser retornados

```
Jogador Ler();
```

Funções e Registros

- Os registros são **passados por valor**

- A função recebe uma cópia do registro

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

```
int SomarGols(Jogador j1, Jogador j2);
```

- Uma alternativa é passar o endereço do registro

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

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

```
int SomarGols(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 Somar(Tempo t1, Tempo t2);
void Mostrar(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 Somar(Tempo t1, Tempo t2);
void Mostrar(Tempo t);
```

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

    Tempo viagem = Somar(dia1, dia2);
    cout << "Total de dois dias: ";
    Mostrar(viagem);

    Tempo dia3 = {4, 32};

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

Funções e Registros

```
Tempo Somar(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 Mostrar(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 &**

```
Imagem Combinar(Imagem * img1, Imagem * img2);
```

```
Imagem a = {10, 20};  
Imagem b = {30, 40};  
Imagem c = Combinar(&a,&b);  
Exibir(&c);
```

```
struct Imagem  
{  
    int altura;  
    int largura;  
    ...  
};
```

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 Combinar(Imagem * img1, Imagem * img2, Imagem * res);
```

```
Imagem a = {10, 20};  
Imagem b = {30, 40};  
Imagem c;  
Combinar(&a, &b, &c);  
Exibir(&c);
```

Funções e Registros

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

struct Imagem
{
    int altura;
    int largura;
};

void Combinar(const Imagem * img1,
              const Imagem * img2,
              Imagem * img3);

void Exibir(const Imagem * img);
```

```
int main()
{
    Imagem foto1 = {10, 20};
    Imagem foto2 = {30, 40};
    Imagem combinada;

    Combinar(&foto1, &foto2, &combinada);
    cout << "Imagem Combinada: ";
    Exibir(&combinada);

    Imagem info = {10, 10};
    Imagem final;
    Combinar(&combinada, &info, &final);

    cout << "Imagem final: ";
    Exibir(&final);
}
```

Funções e Registros

```
void Combinar(const Imagem * img1, const Imagem * img2, Imagem * res)
{
    res->altura = (img1->altura > img2->altura ? img1->altura : img2->altura);
    res->largura = img1->largura + img2->largura;
}
```

```
void Exibir(const Imagem * img)
{
    cout << img->altura << " x " << img->largura << " pixels\n";
}
```

- Saída do programa:

Imagem Combinada: 30 x 60 pixels

Imagem Final: 30 x 70 pixels

Funções e o Tipo `string`

- Variáveis do tipo `string` se parecem bastante com registros
 - Podem ser usadas 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 Inverter(string s);
```

Funções e o Tipo string

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

string Inverter(const string str);

int main()
{
    string planeta;
    cout << "Digite seu planeta favorito:\n";
    getline(cin, planeta);

    cout << "\nSeu planeta é " << Inverter(planeta) << ".\n";
    cout << "Opa! Quiz dizer " << planeta << ".\n";
}
```

Planeta.cpp

Funções e o Tipo string

```
string Inverter(const string str)
{
    string invertida;
    for (int i = int(str.size()) - 1; i >= 0; i--)
        invertida += str[i];
    return invertida;
}
```

■ Saída do programa:

Digite seu planeta favorito:

saturno

Seu planeta é onrutas.

Opa! Quiz dizer saturno.

Resumo

- Ao passar **vetores** para funções:

- Manipula-se a **cadeia original** e não uma cópia
- Para proteger os dados contra alterações pode-se usar **const**

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

- **Registros e variáveis do tipo `string`** são passados por cópia

- Para evitar a cópia de um grande volume de dados é preciso **passar o endereço (&)** ou usar **referências[†]**

```
void Exibir(const Imagem * img)
```