

Tipos Compostos de Dados

# STRINGS

# Introdução

- As **variáveis** e **constantes** armazenam informações
  - Elas ocupam espaço na memória
  - Possuem um tipo
- Os **tipos básicos** armazenam valores:

Inteiros	{	<b>char</b>	ch	=	'W';
		<b>short</b>	sol	=	25;
		<b>int</b>	num	=	45820;
Ponto-flutuantes	{	<b>float</b>	taxa	=	0.25f;
		<b>double</b>	peso	=	1.729156E5;

# Introdução

- Porém, com os tipos básicos não é possível armazenar um **conjunto de informações**
  - Como armazenar as notas de 30 alunos?

```
float n1 = 8.0;  
float n2 = 7.0;  
float n3 = 4.5;  
...  
float n29 = 5.0;  
float n30 = 2.0;
```

Criar 30 variáveis diferentes não é a melhor solução.

- A solução é usar vetores:  
`float notas[30];`

# Introdução

- Tipos compostos de dados armazenam múltiplos valores:
  - Vetores
  - Strings
  - Registros
  - Uniões
  - Enumerações
- Os tipos compostos são coleções formadas a partir dos tipos básicos de dados

# Strings

- As strings são **vetores de caracteres** com uma propriedade especial:
    - O último caractere de toda string é o caractere nulo (escrito `\0`, ele é o caractere de código ASCII 0)
- ```
char dog[5] = {'l', 'a', 't', 'i', 'r'}; // não é string
char cat[5] = {'m', 'i', 'a', 'r', '\0'}; // string
```
- Ambos os exemplos são vetores de caracteres, mas **apenas o segundo é uma string**

# Strings

- O **caractere nulo** tem um papel fundamental em uma string:
  - Todas as funções que trabalham com strings percorrem o vetor até achar o caractere nulo

```
char cat[5] = {'m', 'i', 'a', 'r', '\0'}; // string
char dog[5] = {'l', 'a', 't', 'i', 'r'};  // não é string
```

```
cout << cat; // miar
cout << dog; // latir
```

- Usando **cout** com um vetor de caracteres faz com que ele **exiba lixo da memória até achar um \0**

# Strings

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

```
char penas[10] = "Gaivota"; // caractere \0 está implícito
char peixe[]    = "Sardinha"; // deixe 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

- Um caractere entre aspas duplas não é a mesma coisa que um caractere entre aspas simples

```
char camisa = 'S'; // ok  
char camisa = "S"; // ilegal, tipos diferentes
```

- 'S' corresponde a um único caractere que possui o código ASCII 83
- "S" representa o endereço inicial do conjunto composto pelos caracteres 'S' e '\0'



# Strings

- Uma **constante string** pode ser **muito longa** para caber em uma linha de código
  - C++ **concatena constantes strings** separadas por espaços, tabulações ou novas linhas

// as instruções abaixo são equivalentes

```
cout << "Eu daria tudo para ser uma frase.\n";
```

```
cout << "Eu daria tudo para ser " "uma frase.\n";
```

```
cout << "Eu daria tudo para ser "  
      "uma frase.\n";
```

# Leitura de Strings

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    const int Tam = 15;
    char nome[Tam];           // vetor "vazio"
    char apelido[Tam] = "C++owboy"; // vetor inicializado

    cout << "Olá! Eu sou " << apelido << "! Qual é seu nome?\n";
    cin >> nome;

    cout << "Bem, " << nome << ", seu nome tem " << strlen(nome) << " letras\n";
    cout << "e está armazenado em um vetor de " << sizeof(nome) << " bytes.\n";

    cout << "Sua inicial é " << nome[0] << "...\n";
    apelido[3] = '\0'; // caractere nulo
    cout << "Meu apelido é " << apelido << endl;
}
```

# Leitura de Strings

- A saída do programa:

```
Olá! Eu sou C++owboy! Qual é seu nome?  
Joãozinho  
Bem, Joãozinho, seu nome tem 9 letras  
e está armazenado em um vetor de 15 bytes.  
Sua inicial é J.  
Meu apelido é C++
```

- A atribuição do caractere `'\0'` para a quarta posição do vetor, **encurtou a string** (pelo menos para o cout)

# Leitura de Strings

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

int main()
{
    const int TamVet = 20;

    char nome[TamVet];
    char sobremesa[TamVet];

    cout << "Entre com seu nome:\n";
    cin >> nome;    // lê apenas uma palavra

    cout << "Entre com sua sobremesa favorita:\n";
    cin >> sobremesa;

    cout << "Eu tenho um " << sobremesa;
    cout << " para você, " << nome << ".\n";
}
```

# Leitura de Strings

- A saída do programa:

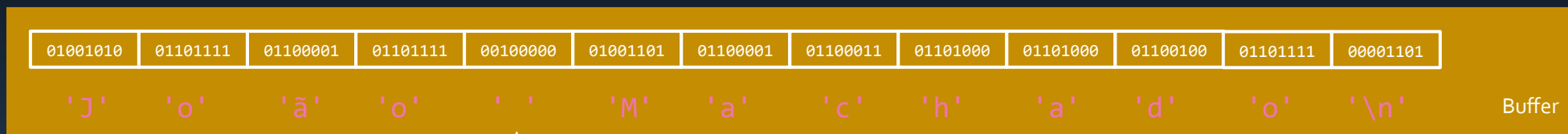
Entre com seu nome:

**João Machado**

Entre com sua sobremesa favorita:

Eu tenho um **Machado** para você, **João**.

- O **fim da entrada de dados** para cin é um espaço em branco, uma tabulação ou uma nova linha



cin encerra a leitura no espaço

# Leitura de Strings

- Como ler uma entrada que contém espaços, como por exemplo "Rio Grande do Norte"?
  - É necessário usar uma função que seja orientada a linhas e não orientada a palavras
- A função `cin.getline()` lê uma linha
  - Até o caractere de nova linha ('\n')

```
char estado[80];  
cin.getline(estado, 80); // lê até 79 caracteres (e insere '\0')
```

# Leitura de Strings

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

int main()
{
    const int TamVet = 20;

    char nome[TamVet];
    char sobremesa[TamVet];

    cout << "Entre com seu nome:\n";
    cin.getline(nome, TamVet);

    cout << "Entre com sua sobremesa favorita:\n";
    cin.getline(sobremesa, TamVet);

    cout << "Eu tenho um " << sobremesa;
    cout << " para você, " << nome << ".\n";
}
```

# Leitura de Strings

- A saída do programa:

Entre com seu nome:

**João Machado**

Entre com sua sobremesa favorita:

**Sorvete**

Eu tenho um **Sorvete** para você, **João Machado**.

- A função `getline()` recebe dois argumentos:

- O **nome do vetor** que receberá a string
- O **limite de caracteres** suportados pelo vetor



# Misturando >> com getline

```
// misturando cin e cin.getline
#include <iostream>
using namespace std;

int main()
{
    cout << "Em que ano sua casa foi construída?\n";
    int ano;
    cin >> ano;

    cout << "Qual é seu endereço?\n";
    char endereco[80];
    cin.getline(endereco, 80);

    cout << "Ano de construção: " << ano << endl;
    cout << "Endereço: " << endereco << "\n";
    cout << "Pronto!\n";
}
```

# Misturando >> com getline

- A saída do programa:

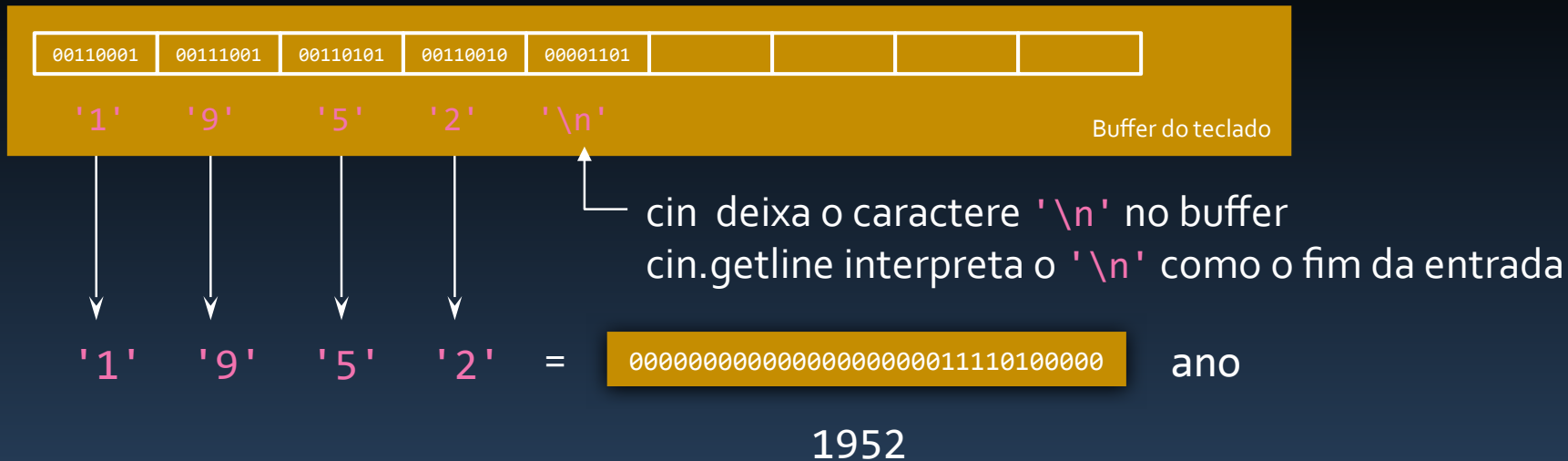
```
Em que ano sua casa foi construída?  
1932  
Qual é seu endereço?  
Ano de construção: 1932  
Endereço:  
Pronto!
```

- O **operador de extração >>** deixa o caractere de nova linha ('\n') no buffer de entrada
  - cin.getline() armazena uma linha vazia na variável

# Misturando >> com getline

- Todo caractere digitado no teclado vai para um espaço temporário de memória chamado de **buffer do teclado**

```
int ano;  
cin >> ano;  
cin.getline(endereco, 80);
```



# Misturando >> com getline

- A solução do problema é **descartar o caractere \n** do buffer:
  - A **função cin.get()**, sem parâmetros, pode ser usada para ler e descartar um caractere

```
cout << "Em que ano sua casa foi construída?\n";  
int ano;  
cin >> ano;  
cin.get();    // caractere \n lido e descartado
```

```
cout << "Qual é seu endereço?\n";  
char endereco[80];  
cin.getline(endereco, 80);
```

# Misturando >> com getline

- A função cin.get() possui **outra versão** que recebe uma **variável tipo char** como argumento
  - Permite armazenar o caractere lido

```
cout << "Em que ano sua casa foi construída?\n";  
int ano;  
cin >> ano;
```

```
char ch;  
cin.get(ch);    // guarda caractere lido em ch
```

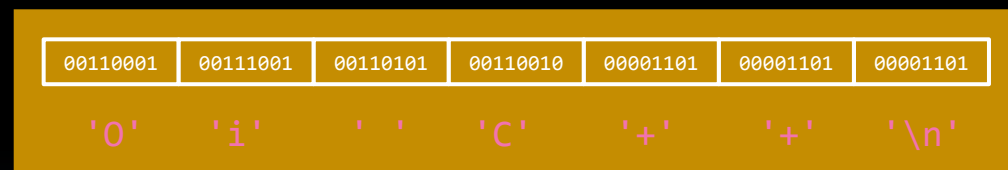
```
cout << "Qual é seu endereço?\n";  
char endereco[80];  
cin.getline(endereco, 80);
```

# Funções de Leitura

- Uma **entrada de texto** pode ser lida:

- Um caractere por vez

```
char ch;  
cin.get(ch);
```



Buffer do teclado

- Uma palavra por vez

```
char palavra[20];  
cin >> palavra;
```

|         |    |
|---------|----|
| ch      | O  |
| palavra | Oi |

- Uma linha por vez

```
char linha[80];  
cin.getline(linha, 80);
```

|       |        |
|-------|--------|
| linha | Oi C++ |
|-------|--------|

# Atribuição e Cópia

- Uma string **não pode ser atribuída** a outra

```
char felino[8] = "Tigre";  
char animal[8];  
animal = felino;  x  // atribuição inválida  
animal = "Tigre"; x  // atribuição inválida
```

- É necessário copiar cada caractere individualmente

```
animal[0] = felino[0];    animal[0] = 'T';  
animal[1] = felino[1];    animal[1] = 'i';  
...                      ...
```

- **Quilizar a função strcpy**  

```
strcpy(animal, felino); // strcpy(destino, origem)  
strcpy(animal, "Tigre"); // strcpy(destino, origem)
```

# Atribuição e Cópia

- Por que essa restrição na atribuição?

```
char felino[8] = "Tigre";  
char animal[8];
```

```
animal = felino;    X // atribuição inválida  
animal = "Tigre";  X // atribuição inválida
```

- Para evitar cópia de vetores:
  - O nome de um vetor é um endereço
  - Uma constante string é um endereço
  - Não é possível mudar os endereços

|   |    |                 |
|---|----|-----------------|
| 0 | T  | 0xCB20 = felino |
| 1 | i  | 0xCB21          |
| 2 | g  | 0xCB22          |
| 3 | r  | 0xCB23          |
| 4 | e  | 0xCB24          |
| 5 | \0 | 0xCB25          |
| 6 |    | 0xCB26          |
| 7 |    | 0xCB27          |

|           |    |                  |
|-----------|----|------------------|
| Constante | T  | 0xD030 = "Tigre" |
|           | i  | 0xD031           |
|           | g  | 0xD032           |
|           | r  | 0xD033           |
|           | e  | 0xD034           |
|           | \0 | 0xD035           |



# Atribuição e Cópia

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

int main()
{
    char felino[20] = "Tigre";
    char animal[20];

    strcpy(animal, felino);
    strcpy(felino, "Jaguar");

    cout << "Felino: " << felino << endl;
    cout << "Animal: " << animal << endl;
}
```

# Atribuição e Cópia

- A saída do programa:

```
Felino: Jaguar  
Animal: Tigre
```

- A função `strcpy` **não verifica o tamanho** do destino
  - O compilador Visual C++ sugere usar `strcpy_s`
    - Porém essa função não é padrão
    - A **mensagem pode ser desabilitada** com um **`#define _CRT_SECURE_NO_WARNINGS`**
      - Deve vir antes do `#include <iostream>`

# 0 Tipo String

- O padrão C++98 introduziu a **classe string** em sua biblioteca

- No lugar de usar um vetor de caracteres para armazenar strings, é possível usar uma variável do **tipo string**

```
string nome; // variável do tipo string
```

- É necessário incluir o **arquivo de cabeçalho** string e ter acesso ao **espaço de nomes** std

```
#include <string>  
using namespace std; // using std::string;
```

# 0 Tipo String

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

int main()
{
    char felino[20] = "jaguar";
    string animal = "pantera";

    cout << "Entre com o nome de dois felinos:\n";
    cin >> felino;
    cin >> animal;

    cout << "Aqui estão os felinos:\n";
    cout << felino << " e " << animal << endl;

    cout << "A terceira letra dos felinos:\n";
    cout << felino[2] << " " << animal[2] << endl;
}
```

# 0 Tipo String

- A saída do programa:

Entre com o nome de dois felinos:

**Tigre**

**Leopardo**

Aqui estão os felinos:

Tigre e Leopardo

A terceira letra dos felinos:

g o

- Uma variável tipo string **pode ser usada da mesma forma** que um vetor de caracteres

# 0 Tipo String

- Inicialização de strings a partir do C++11

```
char tic[20] = "Tic";    // inicialização tradicional de C++
char tac[20] = {"Tac"};  // nova inicialização em C++11
char toe[20]  {"Toe"};   // nova inicialização em C++11
```

```
string tic = "Tic";      // inicialização tradicional de C++
string tac = {"Tac"};    // nova inicialização em C++11
string toe  {"Toe"};     // nova inicialização em C++11
```

- Nem todos os compiladores disponíveis suportam as duas novas formas de inicializar strings
  - Visual C++, g++ e clang++ tem suporte

# 0 Tipo String

- O tipo string **simplifica a atribuição**:

- Não é possível atribuir um vetor a outro

```
char felino[20] = "Tigre";  
char animal[20];  
animal = felino; x           // atribuição inválida de vetores  
strcpy(animal, felino);     // a forma correta de atribuir
```

- Mas é possível atribuir uma string a outra

```
string felino = "Pantera";  
string animal;  
animal = felino;           // atribuição de strings
```

# 0 Tipo String

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    char vAnimal[20];
    char vFelino[20] = "jaguar";
    string sAnimal;
    string sFelino = "pantera";

    strcpy(vAnimal, vFelino);    // copia vetores de caracteres
    sAnimal = sFelino;          // copia strings

    strcat(vAnimal, "ibe");      // adiciona "ibe" ao final do vetor
    sAnimal = sAnimal + " rosa"; // adiciona " rosa" ao final da string

    cout << vAnimal << " contém " << strlen(vAnimal) << " caracteres.\n";
    cout << sAnimal << " contém " << sAnimal.size() << " caracteres.\n";
}
```



# 0 Tipo String

- A saída do programa:

```
jaguaribe contém 9 caracteres.  
pantera rosa contém 12 caracteres.
```

- A string fornece uma **sintaxe mais simples**
- **strcpy** e **strcat** geram problemas se o vetor de destino não for grande o suficiente
- A classe string **redimensiona a string** de destino automaticamente

# 0 Tipo String

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

int main()
{
    char vet[20];
    string str;

    cout << "Comprimento de vet: " << strlen(vet) << endl;
    cout << "Comprimento de str: " << str.size() << endl;

    cout << "Entre com duas linhas de texto: " << endl;
    cin.getline(vet, 20);
    getline(cin, str);

    cout << "Comprimento de vet: " << strlen(vet) << endl;
    cout << "Comprimento de str: " << str.size() << endl;
}
```

# 0 Tipo String

- A saída do programa:

```
Comprimento de vet: 37  
Comprimento de str: 0  
Entre com duas linhas de texto:  
Texto para vet  
Texto para str  
Comprimento de vet: 14  
Comprimento de str: 14
```

- A função **getline** recebe o objeto de entrada

```
getline(cin, str);
```

# Resumo

- Uma **string** é uma sequência de caracteres finalizada pelo caractere nulo `'\0'`
  - Uma string é armazenada em um vetor de caracteres

```
char dica[80] = "Estude C++";  
cin >> dica;
```

- A **função strlen()** retorna o comprimento de uma string

```
char dica[80] = "Estudar em casa é fundamental";  
cout << strlen(dica); // comprimento = 29
```

# Resumo

- A **classe string** fornece uma solução mais simples
  - No lugar de um **vetor** utiliza-se uma variável do **tipo string**

```
#include <string>
using namespace std;    // using std::string;
string nome;            // variável do tipo string
```

- O gerenciamento automático do tamanho da string traz um **custo ao desempenho do programa**
- É importante conhecer a solução usando **vetores**
  - Está **mais próxima** do que realmente acontece na máquina