Introdução a Técnicas de Programação

Manipulação de arquivos

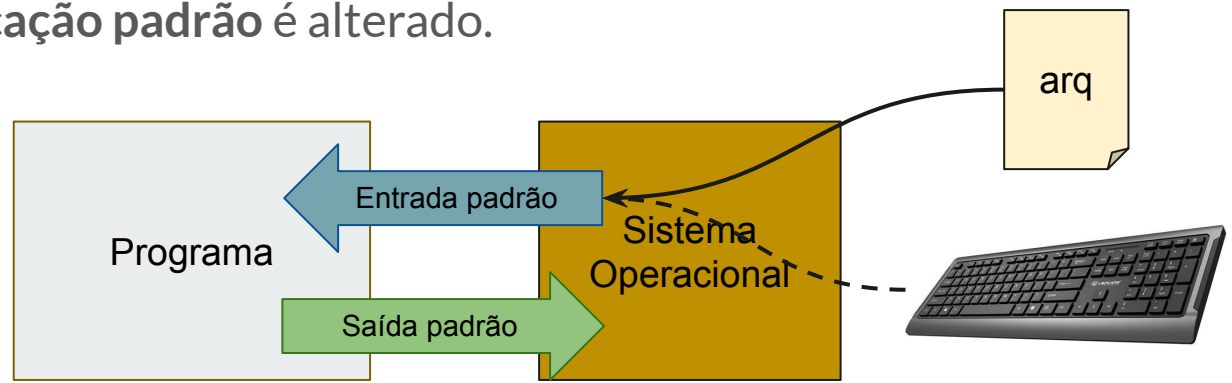
Prof. André Campos
DIMAp/UFRN

Canais de comunicação

Já vimos como ler e escrever arquivos usando o redirecionamento da entrada e saída na linha de comando.

```
$ ./main < entrada.txt
$ ./main > saida.txt
$ ./main < entrada.txt > saida.txt
```

O **canal de comunicação padrão** é alterado.

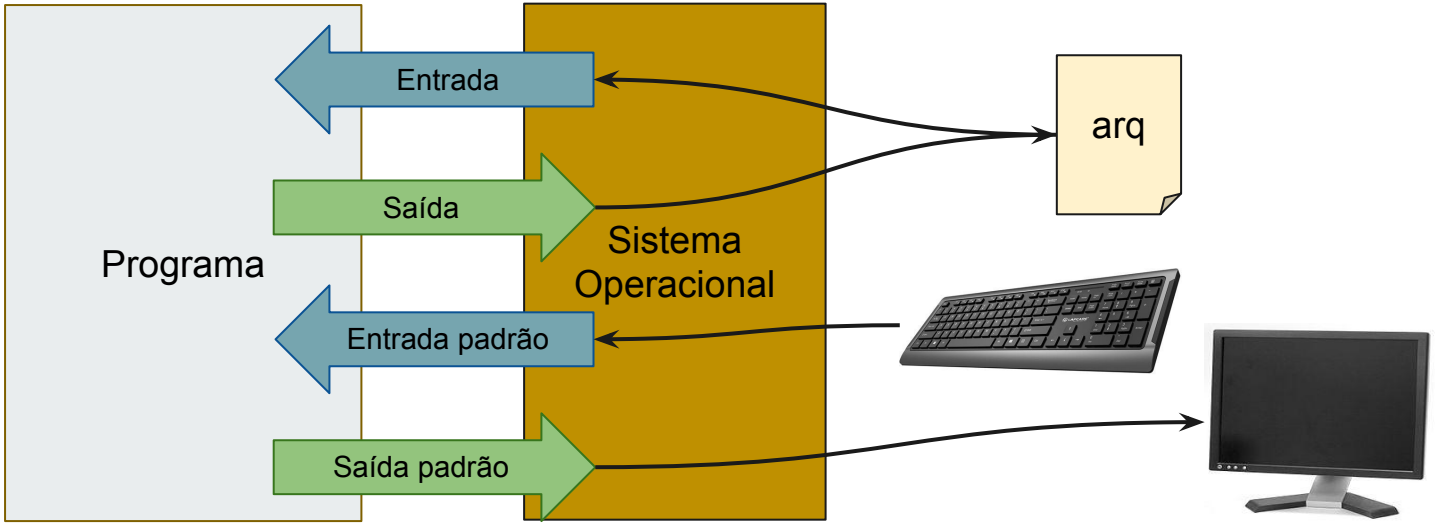


Canais de comunicação

Podemos criar novos canais de comunicação (streams)

Podemos ler e escrever **em arquivos**

- Sem precisar alterar o canal de comunicação padrão!!!

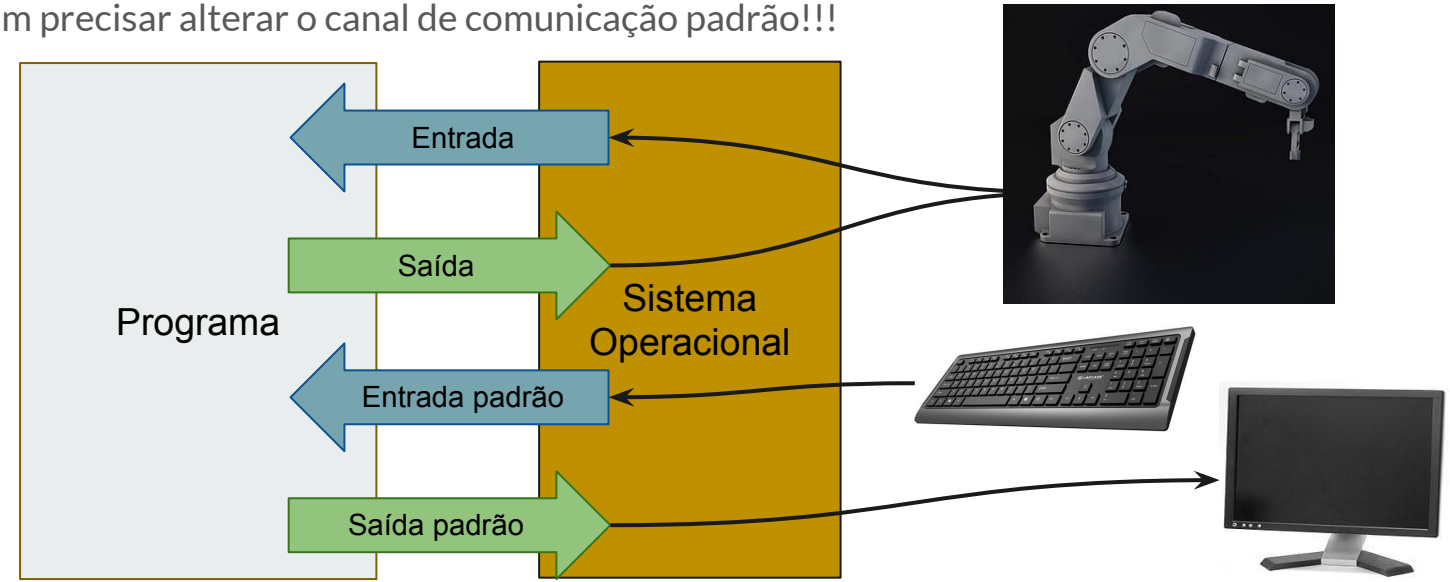


Canais de comunicação

Podemos criar novos canais de comunicação (streams)

Podemos ler e escrever em arquivos ou **outros dispositivos**

- Sem precisar alterar o canal de comunicação padrão!!!

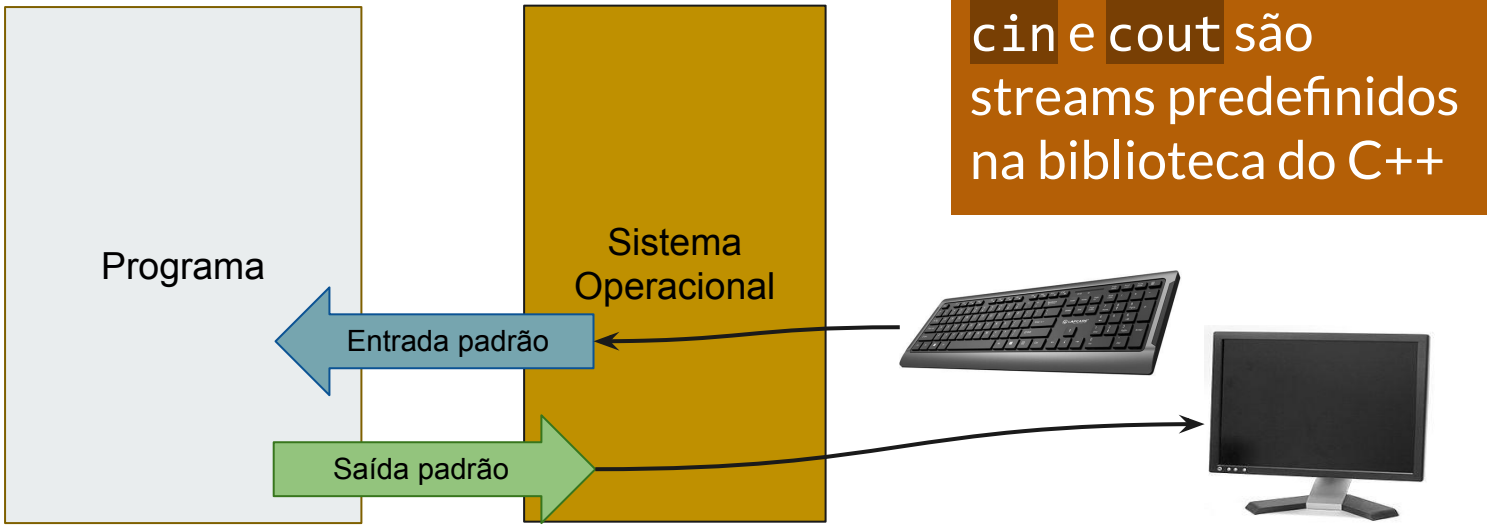


Canais de comunicação

Podemos criar novos canais de comunicação (streams)

Podemos ler e escrever em arquivos ou outros dispositivos

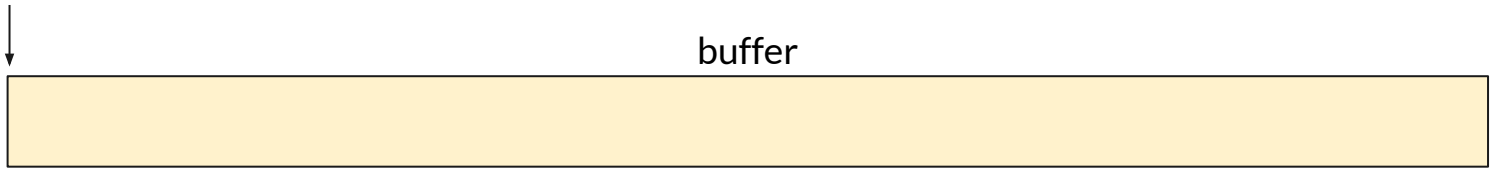
- Sem precisar alterar o canal de comunicação padrão!!!



`cin` e `cout` são streams predefinidos na biblioteca do C++

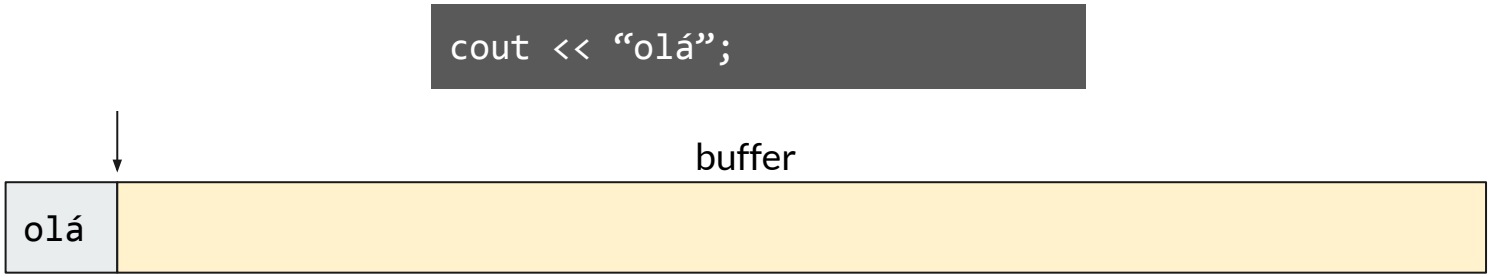
Streams e buffers

Todo stream possui uma área de memória onde os dados são guardados temporariamente, antes de ser enviado para o dispositivo/arquivo.



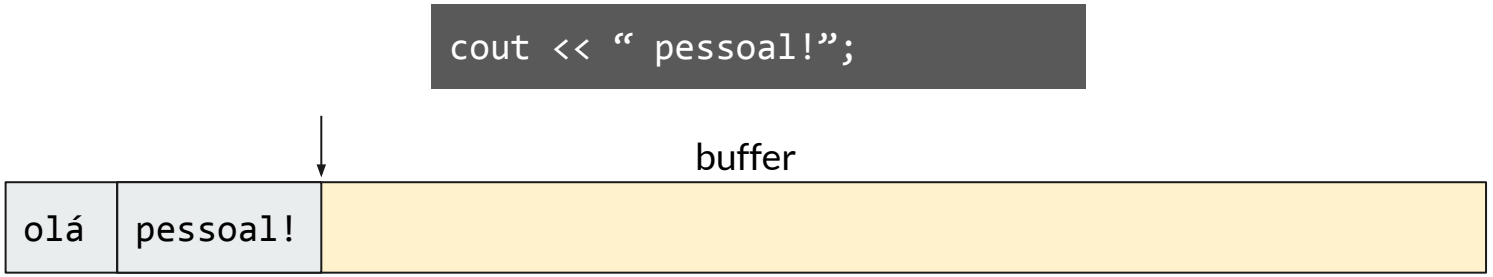
Streams e buffers

Todo stream possui uma área de memória onde os dados são guardados temporariamente, antes de ser enviado para o dispositivo/arquivo.



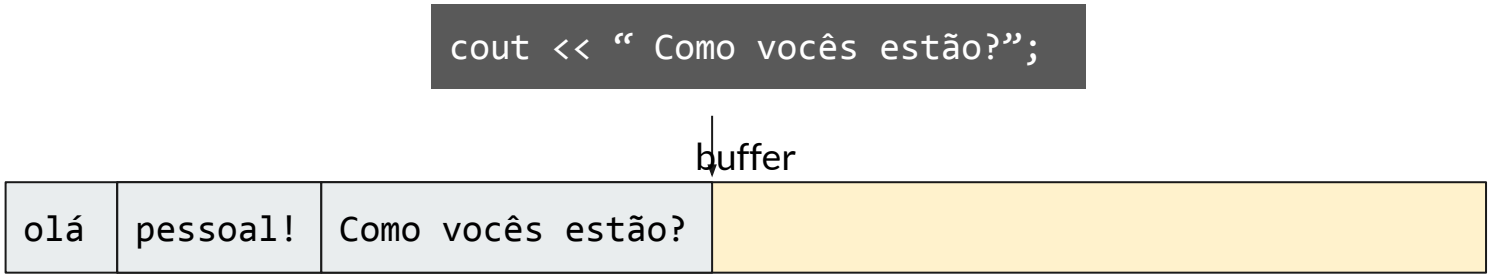
Streams e buffers

Todo stream possui uma área de memória onde os dados são guardados temporariamente, antes de ser enviado para o dispositivo/arquivo.



Streams e buffers

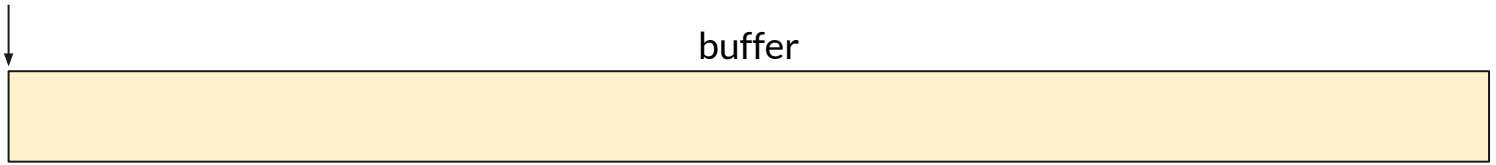
Todo stream possui uma área de memória onde os dados são guardados temporariamente, antes de ser enviado para o dispositivo/arquivo.



Streams e buffers

Todo stream possui uma área de memória onde os dados são guardados temporariamente, antes de ser enviado para o dispositivo/arquivo.

```
cout << endl;
```



Os buffer são eventualmente enviados para os dispositivos (arquivos) quando: 1) o arquivo é fechado; 2) é preenchido; 3) há uma operação de "flush" (`endl`, `flush`…); ou 4) quando a função `sync()` é chamada.

Fluxo de uso dos streams

1. **Abertura**
   - Pode-se indicar o modo de operação: leitura, escrita, texto, binário...

2. **Leitura ou escrita de dados**

3. **Fechamento**
   - **É importante "fechar"** o canal quando não for mais necessário acessar os dados:
     i. É um recurso (memória e processamento) que está sendo utilizado.
     ii. Quando há uma escrita de dados, eles são guardados inicialmente no *buffer* (memória temporária). Ao fechar, o *buffer* é esvaziado e o arquivo atualizado. Se o programa não terminar corretamente, o arquivo pode ficar sem os últimos dados.

Tipos de streams em C++

- **ostream**
  Representa canais de “saída de dados”. Só permite escrever dados.
  Ex: `cout` é um ofstream

- **istream**
  Representa canais de “entrada de dados”. Só permite leitura de dados.
  Ex: `cin` é um ifstream

- **iostream**
  Representa canais em geral, que podem tanto ler quanto escrever.

Streams para arquivos

- **ofstream**
  Canal para criar arquivos ou anexar dados em arquivos existentes.

- **ifstream**
  Canal para ler dados de arquivos existentes.

- **fstream**
  Canal que pode tanto ler quanto escrever em arquivos.

Exemplo em C++

Para usar streams de arquivos, é necessário incluir os cabeçalhos:

```
#include <iostream>
#include <fstream>
```

Ao invés de enviar os dados para `cout`, enviamos para `arquivo`.

Precisamos “fechar” o canal no final.

```
#include <iostream>
#include <fstream>

using namespace std;

int main() {
  ofstream arquivo("hello.txt");
  arquivo << "Hello, world!" << endl;
  arquivo.close();
  return 0;
}
```

Exemplo em C++

Pode-se declarar o stream e efetuar a “conexão” do canal depois.

```cpp
#include <iostream>
#include <fstream>

using namespace std;

int main() {
  ofstream arquivo;
  //...
  arquivo.open("hello.txt");
  arquivo << "Hello, world!" << endl;
  arquivo.close();
  return 0;
}
```

Exemplo em C++

Aconselha-se a testar se houve erro na abertura (`is_open()`):

- Arquivo inexistente
- Arquivo que o usuário não tem permissão para ler ou escrever

Durante o processamento, pode-se testar o estado:

- `bad()`
- `fail()`
- `eof()`
- `good()`

```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main() {
  ofstream arquivo("hello.txt");
  if (arquivo.is_open()) {
    arquivo << "Hello, world!" << endl;
    arquivo.close();
  }
  else {
    cout << "Erro de abertura" << endl;
  }
  return 0;
}
```

Modos de abertura

Pode-se especificar os tipos de operações que faremos no arq.

- `ios::in` - leitura
- `ios::out` - gravação
- `ios::app` - anexar no final
- `ios::trunc` - sobrescrever
- `ios::binary` - binário

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

int main() {
  fstream arquivo("hello.txt",
    ios::in | ios::out | ios::app | ios::binary
  );
  // ...
  arquivo.close();
  return 0;
}
```

Leitura e gravação sem os operadores << e >>

Leitura

- `getline()` - lê dados até um '\n'
- `ignore()` - ignora o próximo byte (char)
- `get()` - várias formas de ler
- `read(char* s, int n)` - lê um bloco de bytes (modo binário)

Gravação

- `put()` - insere um byte (char)
- `write(const char* s, int n)` - grava um bloco de bytes (modo binário)