INF 112: Programação II

Aula 16

- ***Streams*** **em C++ - Arquivos texto**

Fábio R. Cerqueira, UFV, DPI, frcerqueira@gmail.com

➔ **Em C++, bem como em outras linguagens de programação, os fluxos de dados de entrada e saída (E/S) são chamados *streams*. Exemplos:**

- ***File Stream***
- ***Memory Stream***
- ***Audio Stream / Video Stream***
- ***I/O Stream***
- ***Binary Stream***
- ***String Stream***

➔ Os *streams* são, portanto, fluxos sequenciais de dados binários ou em texto, que são enviados para ou recebidos de qualquer dispositivo de E/S: teclado, monitor de vídeo, impressora, arquivo em disco, modem, placa de som etc.

➔ Os *streams* podem ser de entrada, de saída ou de entrada/saída ao mesmo tempo.

➔ Praticamente todo programa que faça algo de útil utiliza algum tipo de *stream*.

- Qualquer *stream,* antes de ser usado efetivamente, deve ser aberto. Isso serve para associar o *stream* (dispositivo lógico) a um dispositivo de E/S, como um arquivo.

- Após a utilização, deve ser fechado. Isso serve para liberar o dispositivo, tornando-o disponível para outro uso. Portanto, é bom estilo de programação fechar o *stream* tão logo que possível.

- Quando um programa termina, todos os *streams* ainda abertos são fechados compulsoriamente pelo sistema.

- Apesar disso, devemos fechá-los explicitamente em nossos programas, liberando o dispositivo utilizado o mais cedo possível.

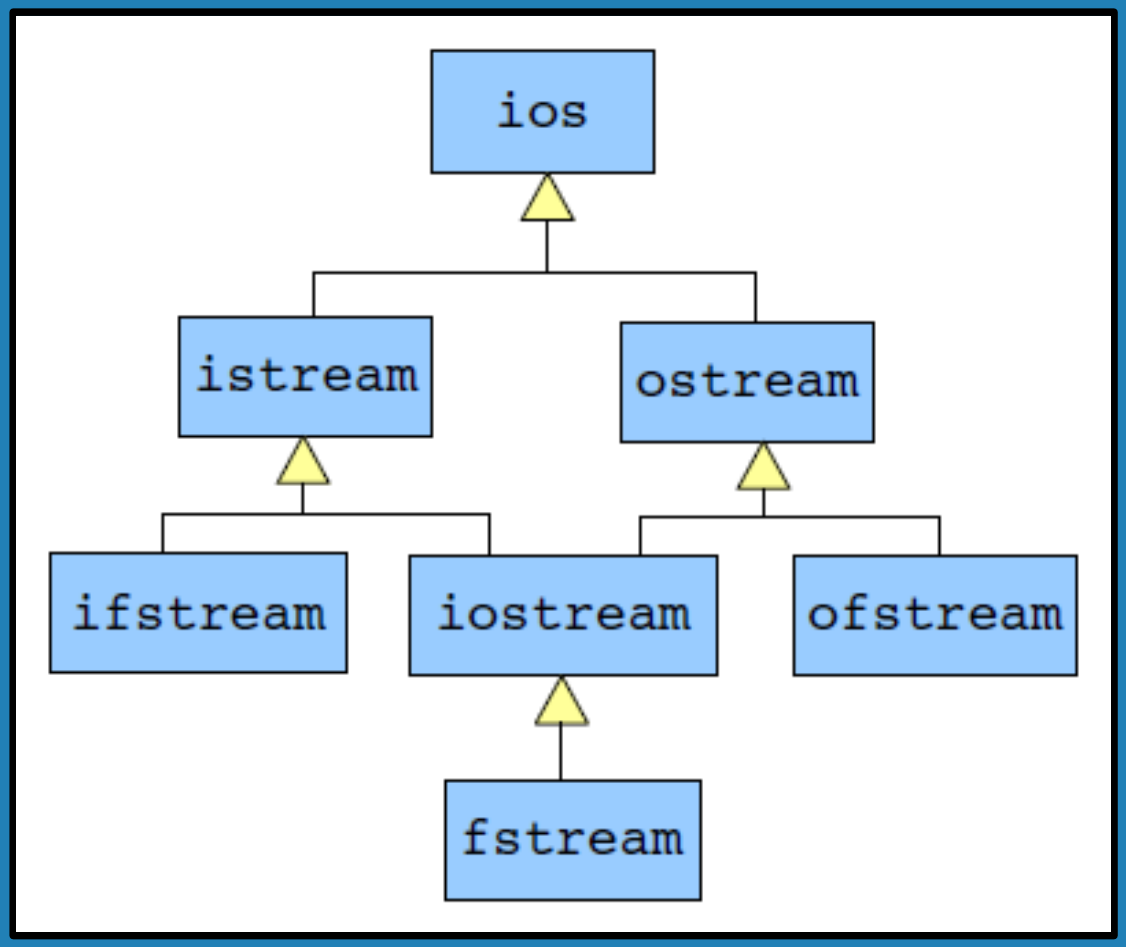
- A abertura é obrigatória, exceto para os *streams* padrões, como *cin* e *cout*, que são abertos e fechados pelo próprio sistema.

Streams em C++

- *Streams* padrões da biblioteca *iostream* do C++:
  - *cin*: *stream* somente de entrada (teclado).
  - *cout*: *stream* somente de saída (monitor de vídeo).
  - *cerr*: *stream* somente de saída utilizado para emitir mensagens de erro.
  - *clog*: *stream* somente de saída utilizado para emitir mensagens de auditoria.

- Informações enviadas via *cout* e *clog* são "bufferizadas". Por exemplo, ao se enviar dados para o vídeo via *cout*, os mesmos são temporariamente armazenados em um *buffer* e depois enviados, de uma só vez, para o vídeo.

- O objeto *cerr*, por sua vez, é um exemplo de uso "não-bufferizado" de *streams*, ou seja, os dados vão imediatamente para o dispositivo de saída.



- Hierarquia de classes para uso de *streams*

Streams em C++ - Arquivos texto

- Para usar *streams* em C++, basta incluir a biblioteca adequada. Exemplo:

```
#include <iostream>
#include <fstream>  // file streams
```

- Para *streams* de arquivo, instanciar objetos (arquivos lógicos) do tipo:

`ifstream`	(arquivos de entrada)
`ofstream`	(arquivos de saída)
`fstream`	(arquivos de entrada/saída)

Streams em C++ - Arquivos texto

➔ Existem vários operadores para manipulação de arquivos (*file streams*) em C++. Seguem alguns:

Operadores	Descrição	Exemplos
`open`	Faz a abertura explícita de um *stream*	`arq.open("notas.txt");`
`>>`	Faz a leitura de um item de um *stream*	`arq >> nota;`
`<<`	Faz a saída de um item para um *stream*	`arq << soma;`
`close`	Fecha explicitamente um *stream*	`arq.close();`
`eof`	Testa a condição de fim dos dados de um *stream*	`arq.eof()`
`fail`	Testa se ocorreu alguma falha na última operação de E/S em um *stream*	`arq.fail()`

- Considere o problema de armazenar 10 temperaturas em *Fahrenheit* num arquivo texto, de modo que o usuário não tenha que entrar com estes valores toda vez que executar um programa que precise dos mesmos.

Streams em C++ - Arquivos texto

```
#include <iostream>
#include <fstream> // biblioteca para streams de arquivo
using namespace std;
main() {
  float TF;
  ofstream arq; // Representará o arquivo de saída
  arq.open("temp_F.txt"); // abre o arquivo
  cout << "Entre com 10 temperaturas em F:\n\n";
  // entra num loop para o usuario digitar os valores
  for ( int i=1; i<=10; i++ ) {
    cout << "Temperatura " << i << ": ";
    cin >> TF; // le valor do teclado
    arq << TF; // envia valor para o arquivo
   if ( i < 10 ) arq << endl;
  }
  arq.close(); // fecha o arquivo
  cout << "\nDados gravados em arquivo.\n\n";
}
```

➔ Considere o problema de ler temperaturas em *Fahrenheit* de um arquivo texto e convertê-las para Celsius.

Streams em C++ - Arquivos texto

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

// Retorna o valor em °C de uma temperatura em °F
float celcius(float F) {return (5.0/9.0)*(F - 32);}

main() {
  float TF;
  ifstream arq; // Representará arquivo com valores em °F
  arq.open("temp_F.txt");
  cout << " F    C\n";
  // Enquanto nao encontrar final do arquivo, le proxima:
  while (! arq.eof()) {  // eof = end of file
    arq >> TF;  // le valor do arquivo
    cout << TF << "    " << celcius( TF ) << endl;
  }
  arq.close();
}
```

Streams em C++ - Arquivos texto

➔ Quando se realiza um *open* em um objeto *ifstream*, utiliza-se o tipo padrão de abertura para tais objetos que é a abertura para leitura.

➔ Quando o arquivo é aberto para leitura, se o mesmo de fato existir, este processo ocorrerá sem problemas e a posição de leitura é colocada automaticamente no início do arquivo.

➔ Se o arquivo não existir, a abertura não é realizada e a tentativa de manipulá-lo perde o sentido (tentativa de usar comandos de leitura, etc.). Daí a importância de se utilizar métodos como o *fail* que retorna *true* se ocorreu falha e *false* caso contrário. Esta observação vale na abertura para escrita (*ofstream*) também. Se não houver espaço suficiente para o arquivo sendo criado, ocorrerá, da mesma forma, um erro de abertura.

➔ Exercício: Acrescente o uso do *fail* nos dois programas anteriores, de modo a cobrir a possibilidade de erro de abertura de arquivo.

- Quando se realiza um *open* em um objeto *ofstream*, utiliza-se o tipo padrão de abertura para tais objetos que é a abertura para escrita.

- Quando o arquivo é aberto para escrita, se o mesmo já existir em disco seu conteúdo é completamente apagado (e a posição de escrita, obviamente, é colocada no início).

- E se quisermos abrir um arquivo para continuar a inserir dados no mesmo a partir de seu fim, sem apagar o que já fora escrito?

➔ O método *open* é sobrecarregado. Há uma versão do *open* em que se pode passar a opção de abertura. Abaixo, alguns dos *flags* para indicar o tipo de abertura de uma *stream:*

Flags	Descrição
ios::in	Abre para operações de entrada (*default* para `ifstream`).
ios::out	Abre para operações de saída (*default* para `ofstream`).
ios::app	Abre para operações de saída e coloca posição de escrita no fim (portanto, as operações são realizadas a partir do final do arquivo).

Streams em C++ - Arquivos texto

```
#include <iostream>
#include <fstream>
using namespace std;
main() {
  float TF;
  ofstream arq;
  arq.open("temp_F.txt", ios::app); // opcao de append
  cout << "Entre com 10 temperaturas em F:\n\n";
  arq << endl; // pula de linha no arquivo
  // loop para o usuario digitar 10 valores
  for ( int i=1; i<=10; i++ ) {
    cout << "Temperatura " << i << ": ";
    cin >> TF;
    arq << TF; // envia valor para o arquivo
    if ( i < 10 ) arq << endl; // pula linha no arquivo
  }
  arq.close(); // fecha o arquivo
  cout << "\nDados gravados em arquivo.\n\n";
 } // O arquivo possui agora + 10 valores. Os dados
   // anteriores não foram perdidos.
```

- Outro exemplo – Diário para as meninas :-)

Streams em C++ - Arquivos texto

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
main() {
  string s; // Novidade aqui também (uso da classe string).
  ofstream arq; // Arquivo de saida
  arq.open("diario.txt", ios::app); // abre arquivo com app
  cout << "Digite seu texto: " << endl << endl;
  do {
    getline( cin,s); // le a linha toda e guarda em s
    if( s !="fim" ) { // se nao for fim
      arq << s; // escreve a linha inteira no arquivo
      arq << endl; // pula linha no arquivo
    }
  } while( s != "fim" ); // enquanto nao for fim
  arq.close(); // fecha o arquivo
  cout << "\nDados gravados em arquivo.\n\n";
}
```

Streams em C++ - Arquivos texto

- A função *getline* serve para ler uma linha inteira (não incluindo o caractere de fim de linha) de uma *stream*.

- Esta função tem dois parâmetros. O primeiro indica de qual *stream* se está lendo. O segundo indica em que *string* a linha lida ficará armazenada.

- Portanto, *getline* pode ser utilizada para ler uma linha de qualquer *stream* de entrada, incluindo arquivos. No exemplo anterior, as linhas foram lidas do teclado. Para ler de arquivo, basta passar para a função o nome do seu *file stream* ao invés de *cin*.

- Veja que o segundo parâmetro da função tem que ser do tipo *string*, não pode ser arranjo de *char*. A seguir, falamos um pouco sobre a classe *string* para os que não estão habituados com a mesma.

Classe *string*

- Há algumas aulas atrás, construímos uma classe `String` com o objetivo de facilitar o uso de *strings* pelo usuário. Às bibliotecas padrões da linguagem C++ adicionou-se uma classe `string` com este mesmo objetivo. Obviamente que esta classe é bem mais completa que a que mostramos. Contém vários métodos e sobrecarga de operadores para facilitar ao máximo o uso de cadeia de caracteres.

- Daremos um exemplo bem simples para mostrar as funcionalidades principais da classe string. Fica a cargo do aluno explorar esta classe para descobrir outros recursos.

- Veja que para utilizar a classe string é necessária a inclusão:

```
#include<string>
```

- Comecemos com uma versão utilizando arranjo de *char*. Depois, veremos outra versão que dá o mesmo resultado, mas utiliza a classe string.

Classe *string* – Exemplo com uso de arranjo de char

```
main() {
  char s1[80], s2[80]; //strings de no maximo 80 caracteres
  int tamS1, tamS2;
  gets(s1); // Funcao para ler linha inteira do teclado
  gets(s2);
  tamS1=strlen(s1); // retorna o tamanho da string
  tamS2=strlen(s2);
  cout << "Tamanho das strings: " << tamS1 << " e " <<
       tamS2 << endl;
  if( !strcmp(s1,s2) ) // comparando strings
    cout << "As strings sao iguais\n";
  else if ( strcmp(s1,s2) < 0 )
    cout << "String 1 vem antes da string 2\n";
  else
    cout << "String 2 vem antes da string 1\n";
  strcat(s1,s2); // concatena segunda string na primeira
  cout << "Apos concatenacao: " << s1 << endl;
  strcpy(s1, s2); // copia segunda string na primeira
  cout << "Apos copiar string 2 na string 1: " << s1 <<
       endl << endl; }
```

Classe *string* – Exemplo com uso da classe string

```
main() {
  string s1, s2; // strings de tamanho qualquer
  int tamS1, tamS2;
  getline(cin, s1); // funcao para ler string do teclado
  getline(cin, s2);
  tamS1=s1.length(); // Metodo para obter tamanho da string
  tamS2=s2.length();
  cout << "Tamanho das strings: " << tamS1 << " e " <<
       tamS2 << endl;
  if( s1 == s2 ) // comparando strings
    cout << "As strings sao iguais\n";
  else if ( s1 < s2 )
    cout << "String 1 vem antes da string 2\n";
  else
    cout << "String 2 vem antes da string 1\n";
  s1 = s1 + s2; // concatena segunda string na primeira
  cout << "Apos concatenacao: " << s1 << endl;
  s1 = s2; // copia segunda string na primeira
  cout << "Apos copiar string 2 na string 1: " << s1 <<
       endl << endl; }
```

Classe *string*

- Podemos ver então que `strcpy` foi substituído por =, `strcat` por +, `strlen` pelo método `length` e `strcmp` pelos operadores de comparação <, > e ==, tornando o uso de cadeia de caracteres mais intuitivo.

- Além disto, a classe já gerencia as questões de memória para o usuário. O mesmo não tem que se preocupar com o tamanho da *string*, nem mesmo com alocação e desalocação de memória.

- Obviamente que há uma contrapartida. A classe `string` utiliza alguma estrutura de dados interna dinâmica para permitir o crescimento flexível da *string*. Isto significa alocação e desalocação intensiva de memória se o objeto *string* em questão sofrer atribuições a todo momento. Ou seja, a performance pode ser significativamente pior comparada ao uso de arranjo de *char* com alocação estática (ou dinâmica por uma única vez).