Streams (fluxos) em C++

Introdução

C++ oferece diversos recursos para acessar arquivos, ou mais genericamente *streams*. O nome *stream* representa um fluxo de entrada ou de saída, e pode ser entendido de diversas formas. Por exemplo, *cout* é um fluxo de saída e *cin* é um fluxo de entrada, ambos associados com o terminal. Da mesma forma, pode-se usar fluxos de saida para *strings*, por exemplo, que permitem compor *strings* como se estivéssemos escrevendo na tela.

String Streams

String stream é um recurso que permite escrevermos em uma *string* como se estivéssemos realizando saída na tela (ex: via *cout*). Dessa forma, é possível utilizar todas as funcionalidades da *iostream* (formatos, alinhamento, precisão numérica, tamanho de campo, etc) em uma *string*.

É útil, por exemplo, para retornarmos informações sobre um objeto sob forma de *string*, especialmente quando é preciso combinar dados numéricos com dados não-numéricos.

Para utilizar o recurso de *string stream*, é necessário primeiramente incluir o *header sstream*. A partir daí, deve-se declarar um objeto do tipo *ostringstream (output string stream)* e utilizá-lo como se fosse uma saída padrão, ou seja, da mesma forma que *cout*:

Finalmente, para se obter a *string* armazenada no objeto *ostringstream* usa-se o método *str*:

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <sstream>

using namespace std;

string converte(float valor)
{
        ostringstream aux;  // Declara o string stream de saída chamado aux
        aux << "Exemplo de saída em string ";  // Escreve em aux
        aux << fixed << setprecision(2) << sqrt(valor); // idem
        return aux.str(); // Retorna a string resultante a partir do stream aux
}

int main()
{
        float v;
        cout << "Digite o valor: ";
        cin >> v;
```

```
        cout << converte(v);
}
```

Note que se o objetivo for apenas concatenar (agrupar) duas ou mais *strings*, não é necessário usar *string stream*: basta empregar o operador de concatenação (+).

É muito comum a criação de um método denominado *toString*, que tem o papel de retornar uma representação do objeto armazendo como uma *string*. Por exemplo:

```
#include <iostream>
#include <sstream>
#include <string>

class Pessoa
{
   private:
      string nome;
      string CPF;
      int idade;

   public:

      Pessoa(string n, string cpf, int i) {
        ...
      }

      ...
      string toString();
      ...
};

...
string Pessoa::toString() {
        // Cria um fluxo de saída em uma string
        ostringstream aux;

        // "Escreve" os dados no fluxo
        aux << nome << " - " << CPF << " - " << idade;

        // Retorna string contida no fluxo
        return aux.str();
}
```

Porém, o mais usual é entender-se fluxos como arquivos físicos, como veremos a seguir.

Streams de Dados (Arquivos)

Quando consideramos streams como arquivos físicos, a sua manipulação se resume em duas etapas:

- A gravação dos dados ou criação do arquivo;
- A leitura dos dados

Gravando em um Stream

Para exemplificar a gravação de dados, criaremos um programa que grava um conjunto de números sorteados em um arquivo texto.

O primeiro passo é incluir o header *fstream*, que contém as classes para manipulação de arquivos físicos (*file streams*). Repare que o programa-exemplo a seguir inclui mais headers, para que seja possível, por exemplo, sortear os números.

```
#include <fstream>      // para usar file streams (ifstream,ofstream)
#include <iostream>     // para usar cin,cout
#include <string>       // para usar string
#include <iomanip>      // para usar manipuladores (setw, right, ...)
#include <cstdlib>      // para usar srand(), rand() e exit()

using namespace std;
```

A seguir, cria-se uma instância de *ofstream*, ou seja, arquivo de saída.

```
int main() {
        // Cria output file stream (ofstream)
        ofstream arqsaida;
```

O próximo passo é abrir o arquivo, usando o modo *ios::out*, que cria o arquivo e abre para escrita. Cuidado: se esse modo for usado em arquivos já existentes, eles serão apagados! O método *is_open()* retorna *false* se o arquivo não está aberto, então usamos para testar se foi possível realizar a operação.

```
        // Cria e abre arquivo
        arqsaida.open( "teste.txt" , ios::out );

        // Se houver erro, sai do programa
        if (!arqsaida.is_open())
                return 0;
```

Neste ponto, chamamos a função *srand*, usando o resultado da função *time* como parâmetro: esta última retorna o tempo atual, e *srand* é utilizada para incializar o gerador de números aleatórios (o que chamamos de *semente*). Como se emprega o tempo atual do sistema, o gerador sempre será inicializado com uma semente diferente, garantindo números "mais" aleatórios.

```
        // Gera a semente aleatória
        srand(time(0));
```

Agora podemos começar a gravar os dados. O nosso arquivo *teste.txt* conterá um cabeçalho (uma linha com um texto qualquer), e uma sequência de 10000 números aleatórios (obtidos com *rand*). Repare que a forma de gravar é idêntica à maneira como escrevemos informações na tela via *cout*.

```
        cout << "Gerando dados..." << endl;

        // Grava o cabeçalho
        arqsaida << "Cabecalho do arquivo" << endl;

        // Agora grava os 10000 registros numéricos
        for (int i = 0; i < 10000; i++) {
                int num = rand() % 10000;
                arqsaida << i << setw(10) << num << endl;
                if(arqsaida.fail()) {
                        cout << "Erro fatal!" << endl;
                        exit(1);                // Aborta programa
                }
        }
```

Veja também neste ponto que empregamos o que se denomina *modificador de formato*: **setw(10)**, que tem a função de ajustar a largura do próximo dado a ser enviado. Nesse caso, ele serve para escrever cada valor com exatamente 10 caracteres, alinhados à direita (padrão para números inteiros). A cada dado gravado, também verificamos se não houve nenhum erro, através da chamada a *fail()*, que retorna *true* se ocorreu algum erro de entrada ou saída.

Finalmente, o último (e importante) passo é fechar o arquivo, de forma que os dados que ainda estiverem na memória sejam enviados para o meio físico. Se essa operação não for realizada, os últimos dados gravados no arquivo podem ser perdidos!

```
        cout << "Fechando o arquivo..." << endl;
        arqsaida.close();

        return 0;
}
```

Experimento

Analise com atenção o programa acima, tentando compreender todo o seu funcionamento. Compile e execute o programa, e verifique que este gerou corretamente o arquivo *teste.txt* no diretório corrente.

Lendo de um Stream

Como utilizaremos os mesmos headers do programa anterior, estes não serão novamente descritos aqui. O primeiro passo então é criar um objeto *ifstream*, que representa o arquivo sendo lido, e associá-lo ao arquivo físico com a chamada *open*.

Repare que o processo é similar à gravação, porém emprega-se o parâmetro *ios::in* para indicar leitura e não gravação.

```
...

int main() {
        // Cria input file stream (ifstream)
        ifstream arq;

        cout << "Abrindo arquivo texto..." << endl;

        // Abre arquivo
        arq.open( "teste.txt" , ios::in );

        // Se houver erro, sai do programa
        if (!arq.is_open())
                return (0);
```

A seguir, é preciso ler primeiramente o cabeçalho. Note que este deve ser lido com *getline*, uma vez que é uma string com espaços em branco:

```
        // Lê cabeçalho
        string cabecalho;
        getline(arq,cabecalho);

        // Exibe cabeçalho na tela
        cout << cabecalho << endl;
```

Uma vez lido o cabeçalho, sabe-se que os próximos registros são compostos de dois números: um contador e o valor armazenado. Basta então fazer uma repetição, que terminará quando não houver mais dados no arquivo. O método *good* retorna *false* quando algo diferente acontecer, por exemplo, um erro ou o próprio final do arquivo. Repare que também só exibimos o dado lido na tela, se *fail* retornar *false*.

```
        // Agora, lê n registros numéricos
        do
        {
                int num, valor;
                arq >> num >> valor;
                if(!arq.fail()) {
                        cout << num << "\t" << valor << endl;
                }
        } while(arq.good());
```

Neste ponto, o laço pode ter terminado por dois motivos: ou houve um erro, ou o arquivo terminou. No primeiro caso, o método *bad* retorna *true*, e no segundo caso, o método *eof* retorna true. Então, uma situação de erro é quando *bad* retornou true, ou *eof* retornou *false*:

```
        if(arq.bad() || !arq.eof()) {
                cout << "Erro fatal!" << endl;
                exit(1);                // Aborta programa
        }
```

Por fim, fechamos novamente o arquivo e encerramos o programa.

```
        cout << "Fechando o arquivo..." << endl;
        arq.close();

        return 0;
}
```

Experimento

Compile o programa acima, e verifique que ele é capaz de ler o arquivo-texto gerado pelo programa anterior. Dica: como há 10000 registros, você pode empregar alguns comandos do *Linux* para facilitar a depuração: por exemplo, redirecionando a saída do programa para o comando *more* permite que você visualize uma tela de cada vez.

```
./read | more
```

Outra dica é redirecionar para os comandos *head* e *tail*: o primeiro mostra as n primeiras linhas da saída, e o último mostra as n últimas linhas da saída (o valor de n é configurável com um parâmetro).

```
./read | head
```

```
./read | tail
```

Há muito mais detalhes do que foi apresentado nesta visão simplificada. Você pode encontrar uma descrição completa dos métodos de E/S em C++ no Guia de Referência, disponível aqui.

Sugestões:

- Explore os demais modificadores de formato (veja o item *flags*, depois clique em *io stream format flags* para ver a descrição de todos). Verifique quais modificadores afetam quais tipos de dados e experimente gravar ints, floats, etc.
- Por que usamos *getline* ao invés de >> para ler a string ? E o que aconteceria nesse caso ? Troque no código e veja o resultado.

Exercícios

Implemente um sistema para contabilização de votos:

- Ler os dados dos candidatos do arquivo candidatos.txt, armazenando as informações na memória;

- Ler os dados da votação de cada urna (arquivos urna[1-4].txt), acumulando os votos de cada candidato;
- Exibir na tela o relatório da votação:
    - Candidato mais votado;
    - Candidato menos votado;
    - Percentual de votos do candidato mais votado em relação ao total;

Zip com arquivos de dados para o exercício

O formato dos arquivos (por linha) é o seguinte:

- Arquivo candidatos.txt: `número_do_candidato nome_do_candidato nome_partido`. Ex:

```
1 Roubalino PDR
2 Estelionatino PDR
3 Robaldo PD
```

- Arquivo urna[1-4].txt: número do candidato para este voto. Ex:

```
5
10
7
3
6
...
```

Dicas: leia números e strings com >>. Não é necessário usar *getline*, pois não há strings com mais de uma palavra. Defina uma classe para representar um candidato. Crie um vetor de candidatos para contabilizar os votos.