Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Linguagem de Programação I • DIM0120
◁ Exercícios sobre *Streams* em C++ ▷
6 de agosto de 2019

Apresentação

Este documento[1] aborda o o uso de **fluxo** (*stream*) em C++. Nas seções seguintes são apresentadas breves explicações sobre fluxos, focando no seu uso para realizar entrada e saída de dados a partir do terminal, *string* e arquivo texto.

O texto ainda descreve uma sequência de exercícios organizados em grau crescente de complexidade, de maneira a usar fluxo das mais variadas formas.

Sumário

[1] Baseado em [1].

1 Introdução

Fluxo em C++ é uma importante estrutura de dados projetada para suportar o *fluxo de informação* a partir de uma *fonte de dados* especificada. A principal forma de se comunicar com um fluxo é através dos **operadores de inserção** `<<` e do **operador de extração** `>>` de informações de um fluxo.

Para mais informações sobre fluxo, recomenda-se visitar http://en.cppreference.com/w/cpp/io, que oferece uma visão geral de referência sobre fluxos, e http://www.cplusplus.com/doc/tutorial/basic_io, onde você pode encontrar um tutorial básico sobre manipulação de entrada/saída de dados em C++.

Métodos de Manipulação Associados a Fluxos

Antes de apresentarmos exemplos de manipulação de dados via fluxos, é necessário introduzir brevemente alguns métodos associado a fluxos. Estes métodos oferecem operações básicas sobre fluxos. Para facilitar o entendimento, os métodos estão organizados em tabelas diferentes. A Tabela 1 apresenta alguns métodos comuns a **todos os fluxos**, independente de serem de entrada ou de saída, ou de estarem ou não associados a arquivo, *string* ou terminal. Para uma lista mais completa, visite http://en.cppreference.com/w/cpp/io/basic_ios.

A Tabela 2 resume os métodos que podem ser usados por qualquer fluxo de entrada de dados (associado a arquivos ou não, como o `std::cin`). Já a Tabela 3 resume os métodos que podem ser usados por qualquer fluxo de saída de dados.

Método	Descrição
$fluxo$`.fail()`	Retorna `true` se o fluxo estiver em estado de falha. Esta condição ocorre quando se tenta ler dados além do fim de um arquivo ou os dados associado ao fluxo foram corrompidos.
$fluxo$`.eof()`	Retorna `true` se o fluxo apontar para o fim do arquivo. Este método normalmente é usado após uma chamada `fail`, de maneira a identificar se a falha do fluxo aconteceu porque já se chegou ao fim dos dados ou por outro motivo de erro.
$fluxo$`.clear()`	Reinicializa os bits associados com um fluxo. Este método normalmente é chamado sempre que precisamos reutilizar um fluxo depois que ele falhou.
`if (`$fluxo$`) ...`	Verifica se um fluxo é válido após uso. É uma chamada equivalente a `if ( !`$fluxo$`.fail() )`.

Tabela 1: Métodos de manipulação associado a todas os tipos de fluxo.

Método	Descrição
`fluxo >> var`	Lê um dado formatado em var. O formato é controlado pelo tipo de dado da variável e pelo uso de manipuladores.
`fluxo.get(ch)`	Lê o próximo caractere na variável ch, a qual é passada por referência. O valor retornado é o próprio fluxo, com o bit `fail` ligado.
`fluxo.get()`	Lê o próximo caractere do fluxo. O retorno do método é um inteiro, que pode ser usado para detectar o caractere de fim-de-arquivo, representado pela constante `EOF`.
`fluxo.unget()`	Retrocede o ponteiro interno do fluxo, de maneira que o caractere já lido poderá ser lido novamente por uma chamada a `get`.
`getline(fluxo, str)`	Lê a próxima linha do $fluxo$ na variável *string* str. A função `getline` retorna o próprio fluxo, o que facilita o teste de fim-de-arquivo.

Tabela 2: Métodos de manipulação suportados por todas os fluxos de entrada.

Método	Descrição
`fluxo << expressão`	Escreve dado formatado no fluxo de saída. O dado formatado é controlado pelo tipo resultado da 'expressão' e por qualquer manipulador sendo utilizado.
`fluxo.put(ch)`	Escreve o caractere ch no fluxo de saída.

Tabela 3: Métodos de manipulação suportados por todas os fluxos de saída.

2 Usando Fluxo Associado ao Terminal

Um dos primeiros contatos com fluxos ocorre quando utilizamos os objetos `std::cout` e `std::cin` para imprimir e ler dados do terminal, respectivamente. Normalmente utilizamos os operadores `<<` com o `cout` e `>>` com o `cin`. Estes operadores suportam as chamadas interações *formatadas*, as quais serão descritas a seguir.

2.1 Saída Formatada

Os operadores de fluxo do C++ permitem o encadeamento de operações, de maneira a facilitar a concatenação de várias operações, como no exemplo abaixo. Note o uso do **manipulador** de fim de linha, `std::endl`.

```
cout << "O total da nota fiscal é de " << valNota << " reais." << endl;
```

Os manipuladores são expostos através do cabeçalho `<iostream>`, embora alguns manipuladores que requerem parâmetros são expostos por outro cabeçalho, `<iomanip>`. Outra característica

importante de um manipulador é que seu efeito sobre o fluxo pode ser *transiente* ou *persistente*. A Tabela 4 apresenta uma lista de alguns manipuladores importantes para os exercícios deste documento. Para uma lista mais compreensiva, visite http://en.cppreference.com/w/cpp/io/manip.

Manipulador	Descrição
`endl`	Insere um fim-de-linha no fluxo de saída e se certifica que os dados são enviados para o fluxo.
`setw(`n`)`	Define n como sendo a largura do próximo campo a ser enviado para o fluxo. Propriedade *transiente*.
`setprecision(`$digits$`)`	Define $digits$ como sendo o número de dígitos que devem aparece após o ponto decimal, quando o modo `scientific` ou `fixed` tiver sido ativado anteriormente. Propriedade *persistente*.
`setfill(`ch`)`	Define ch como sendo o caracteres de preenchimento para um campo no fluxo. O valor default é espaço em branco. Propriedade *persistente*.
`left`	Alinha o valor de um campo à esquerda, ou seja, o caractere de preenchimento, se usado, ficará à direita do valor do campo. Propriedade *persistente*.
`right`	Alinha o valor de um campo à direita, ou seja, o caractere de preenchimento, se usado, ficará à esquerda do valor do campo. Propriedade *persistente*.
`scientific`	Valores em ponto flutuante serão exibidos com notação científica. Propriedade *persistente*.
`fixed`	Valores em ponto flutuante serão exibidos com ponto fixo. Propriedade *persistente*.

Tabela 4: Lista de alguns manipuladores importantes para o trabalho.

2.2 Exemplo #1: Uso de manipuladores para formatar saída

O código a seguir demonstra o uso de alguns manipuladores da Tabela 4 e o resultado obtido, no final da listagem. Em particular, note o uso de definição da largura de um campo, da indicação do caractere de preenchimento de um campo, o controle de precisão e formato de exibição de números reais.

```
1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
5 int main() {
6     const double N_PI = 3.14159265358979323846;
7     const double N_GRAVITY = 9.80665; // em m/s2
8
```

```
    // Imprimindo 2 valores em 3 precisoes diferentes.
    for ( auto prec(2) ; prec < 5 ; ++prec ) {
        cout << setw(40) << setfill('-') << "" << endl; // Imprimir linha
        cout << setfill(' '); // Preencher com espaco em branco.
        cout << fixed << setprecision(prec) << "[" << setw(12) << N_PI << "]" << endl;
        cout << setfill('*'); // Preencher com '*'.
        cout << scientific << setprecision(prec) << "[" << setw(12) << N_GRAVITY << "]" << endl;
    }
    return EXIT_SUCCESS;
}
/* Saida produzida:
----------------------------------------
[        3.14]
[****9.81e+00]
----------------------------------------
[       3.142]
[***9.807e+00]
----------------------------------------
[      3.1416]
[**9.8066e+00]
*/
```

2.3 Exercício 1: Criar função `printHeader`

Utilize seus conhecimentos para criar uma função denominada `printHeader` que gera e imprime na saída padrão o cabeçalho de uma tabela. Os rótulos de cada coluna do cabeçalho devem ser especificados no próprio programa (i.e. *hard-coded*) através de um vetor de *strings*. Confira abaixo o esqueleto geral do programa:

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

/**!
 *  Print the table's header to the terminal (std::cout).
 *  @param fields_ Reference to the list of header fields to be printed.
 */
void printHeader( const std::vector< std::string > & fields_ ) {
    // TODO
}

int main() {
    std::vector< std::string > fieldNames = { "ITEM", "QUANTIDADE", "VALOR UNITARIO" };
    printHeader( fieldNames ); // Imprimir cabecalho da tabela.
    return EXIT_SUCCESS;
}
```

Cada campo deve ser delimitado por linhas verticais `'|'` e conter um espaço em branco de cada lado. Em resumo, o cabeçalho deve ser gerado automaticamente por meio de laços e dos comandos de manipulação de fluxo de maneira a produzir a seguinte saída:

```
+------+------------+----------------+
| ITEM | QUANTIDADE | VALOR UNITARIO |
+------+------------+----------------+
```

Importante: Seu código deve ser desenvolvido de tal maneira que se for necessário acrescentar um campo extra no cabeçalho da tabela, bastaria incluir o novo nome em `fieldNames`, **sem ser necessário alterar a função** `printHeader`.

2.4 Entrada formatada

Para ler dados do terminal, fazemos uso do objeto fluxo `std::cin` e do operador de extração `>>`. Sua utilização é bastante simples, porém com algumas limitações consideráveis.

A forma trivial de ler um valor digitado pelo usuário para uma variável de programa seria:

```
int val;
float x, y;

cout << "Por favor, digite um valor e pressione [enter]: ";
cin >> val;
cout << "Indica as coordenadas de um ponto no espaco (x,y): ";
cin >> x >> y;
```

2.5 Limitações da entrada formatada

São duas as principais limitações deste método. A primeira delas é que se o usuário digitar um valor cujo tipo seja diferente da variável esperada, a entrada é ignorada e nenhum erro é gerado. Por exemplo, se o programa espera um inteiro como em `cin >> val` listado anteriormente e o usuário digita '$16a$' por engano, o valor da variável será '16', o caractere 'a' é ignorado e nenhuma mensagem de erro é gerada!

A segunda limitação é que se tentarmos ler uma variável *string*, desta forma

```
string s;
cin >> s;
```

e o usuário digitar `"Esta eh uma string."` a variável `s` receberá apenas o valor `"Esta"`, parando a leitura na primeira ocorrência de um espaço em branco, sinal de tabulação, etc. Para conseguirmos ler uma *string* tradicional, com espaços em branco, será necessário usar um outro método de leitura, como por exemplo a função `getline`, explicada na Seção 3.2.

2.6 Entrada formata redirecionada

Outra ocasião em que a leitura de dados formatados via `std::cin` pode ser interessante ocorre em problemas computacionais no qual se deseja ler uma quantidade indefinida de dados de um certo tipo. Por exemplo, em uma das listas de programação foi pedido que você recebesse um número indefinido de inteiros, para então classificar este número em faixas de valores $[0; 25)$, $[25; 50)$, $[50; 75)$, $[75; 100)$. Neste caso, você pode fazer uso de uma estratégia simples

```
int val;
while( cin >> val ) {
    if ( val >= 0 and val < 25 ) // Classificar 'val' na primeira faixa.
    ...
}
```

Isso é possível apenas porque o comando de extração `>>` retorna o próprio fluxo como resposta e o fluxo pode ser convertido automaticamente para um tipo boleano, indicando se a leitura foi bem sucedida ou não. Para finalizar a entrada, é necessário enviar o comando de fim de arquivo, ou `Ctrl+D`, de maneira a fazer o fluxo se fechar e, portanto, sair do laço.

A vantagem desta simples estratégia é poder redirecionar um arquivo texto de entrada de dados para o programa, sem necessariamente utilizar abertura de arquivo (explicada na Seção 3). Para isso, basta usar

```
[prompt] .\programa < arquivo.txt
```

onde `programa` representa o nome do programa executável e `arquivo.txt` é um arquivo texto que contém todas as entradas já digitadas.

3 Usando Fluxo Associado a Arquivos

Muitas vezes em aplicações mais complexas precisamos abrir um ou mais arquivos simultaneamente para gravar ou ler dados. Portanto, precisamos desenvolver programas mais versáteis, capazes de ler e gravar dados a partir de arquivos. Os arquivos podem ser gravados no formato **binário** ou **texto**. Neste documento, contudo, abordaremos apenas tratamento de arquivos no modo texto, embora os arquivos binários sejam manipulados similarmente.

Um arquivo, de entrada ou de saída, será tratado por um programa como um fluxo, ou seja, a forma de inserção e remoção de dados é similar a utilizada anteriormente. Uma estratégia geral para ler de ou gravar dados em um arquivo, pode ser definida como:

1. *Declarar um fluxo associado a um arquivo.* Existem dois tipos de fluxo principais[2] que podemos associar com arquivo para ler ou escrever. São eles:

   ```
   ifstream inFile;    // Fluxo de leitura de dados de arquivo.
   ofstream outFile;   // Fluxo para escrita de dados em arquivo.
   ```

2. *Abrir o arquivo.* Antes de acessar o fluxo, é necessário associá-lo a um arquivo existente. Para isso utilizamos o comando `open`. Considere um exemplo no qual desejamos abrir o arquivo `tabela.txt` para leitura. A comando correspondente seria

   ```
   inFile.open("tabela.txt");    // Abrindo e associando fluxo ao arquivo.
   ```

   Neste caso, para que o comando anterior funcione corretamente, estamos assumindo que o arquivo `tabela.txt` (i) *existe* e (ii) está *na mesma pasta em que o programa está sendo executado*. Se o arquivo estiver localizado em outra pasta, é necessário informar o caminho para o arquivo juntamente com o nome do arquivo.

   Alternativamente, se o nome do arquivo estiver armazenado em uma variável *string*, é necessário converter a variável para o estilo-C de representação de *strings*. Isso é feito por meio de método `c_str()` da classe `std::string`, como em

   ```
   std::string fileName = "/userX/data/table.txt";
   inFile.open(fileName.c_str());    // Abrindo arquivo no caminho indicado.
   ```

[2] Também existem outros tipos de fluxo, que permitem a leitura e gravação de dados simultaneamente.

3. *Transferir os dados.* Neste ponto o programa deve interagir com o fluxo associado ao arquivo e utilizar seus operadores e métodos para ler dados para o programa ou enviar dados do programa para o arquivo. A estratégia de leitura e gravação é variada, podendo ser orientada a um caractere por vez, um dado por vez, ou mesmo uma linha inteira por vez.
4. *Fechar o arquivo.* Nesta fase, indicamos ao sistema de arquivos que não mais precisamos ter acesso ao arquivo. Portanto, interrompemos a conexão entre o fluxo e o arquivo por meio do método `close()`, como em

```
inFile.close();
```

3.1 Exemplo #2: Lendo dados de arquivo orientado a caracteres

No exemplo apresentado a seguir, foi utilizado uma função para pedir ao usuário que forneça o nome de um arquivo texto de entrada. Se o arquivo existir, o programa abre o arquivo e o associa a um fluxo de entrada passado para a função por referência. Se o programa não conseguir abrir o arquivo indicado pelo usuário, a função indica o problema e pede um novo nome de arquivo.

Após a abertura do arquivo e sua associação a um fluxo de entrada, o programa *lê um caractere por vez* do arquivo de entrada e o imprime na tela (terminal). A leitura do caractere é repetida até que o arquivo de entrada tenha sido completamente lido (e impresso). Note que a função usada para extrair informação do fluxo de entrada foi o método `get(`ch`)` (ver Tabela 2).

```
1  #include <iostream> // std::cout, std::cin
2  #include <fstream>  // std::ifstream
3  #include <string>   // std::string
4
5  using namespace std;
6
7  /**!
8   *   Asks user for the name of an input file, tries to open
9   *   the file and assigns it to the ifstream passed as reference.
10  *   If the input file name provided by the user does not exist,
11  *   the function keeps asking the user to try again.
12  *
13  *   @param ifs_ Reference to the input file stream we want to open.
14  *   @param msg_ Message to show to the user, asking for a file name.
15  *   @return A string with the input file name provided by the user.
16  */
17 string askUserFileName( ifstream & ifs_, string msg_ = "" ) {
18     string fileName;
19     while( true ) {  // Keep asking the file name until a valid name is provided.
20         cout << msg_;                        // Show message.
21         getline( cin, fileName );            // Get the file name.
22         ifs_.open( fileName.c_str() );       // Try to open the file.
23         if( ! ifs_.fail() ) return fileName; // Job done!
24         fileName.clear();                    // Clear file name, try again...
25         cout << "Unable to open the file. Try again.\n";
26         if ( msg_ == "" ) msg_ = "Enter file name: ";
27     }
28 }
29
30 int main() {
```

```
31      ifstream ifs;
32      askUserFileName( ifs, "Please, enter a file name: " );
33      char ch;                  // Stores the char we read from the input file.
34      while( ifs.get( ch ) ) { // Keeps reading until EOF.
35          cout.put( ch );       // Just print the character on the screen.
36      }
37      ifs.close();
38
39      return EXIT_SUCCESS;
40  }
```

3.2 Exemplo #3: Lendo dados de arquivo orientado a linha

Como os arquivos texto são organizados em linhas, um boa estratégia de leitura de dados seria ler uma linha por vez, com a função `getline(`$fluxo$`,`str`)`, apresentada anteriormente na Tabela 2.

Esta função lê uma linha inteira do fluxo de entrada para a str, ignorando o `'\n'`. Esta função também retorna o fluxo, o que possibilita colocar a chamada dentro de um condicional para verificar se a leitura foi realizada com sucesso.

No Exemplo #2, se quisermos ler linhas por vez, ao invés de caractere, precisaríamos substituir o laço central da função principal por

```
string line;
while( getline( ifs, line ) ) {
    cout << line << endl;
}
```

3.3 Exemplo #4: Leitura formatada de dados de um arquivo

Além da leitura de um caractere ou de uma linha por vez, é possível usar os operadores de inserção e extração formatadas em fluxo, respectivamente, `<<` e `>>`.

Por exemplo, considere o problema descrito anteriormente na Seção 2.6, no qual precisávamos ler vários inteiros de um arquivo de entrada para classificá-los como pertencendo ou não a um conjunto de 4 faixas de valores.

Para este caso, abrimos o arquivo de entrada com um `ifstream` no lugar do `cin` e usamos o operador `>>` da mesma forma para extrair inteiros do fluxo associado ao arquivo. Como já mencionado anteriormente, o `>>` retorna o próprio fluxo, o qual vai indicar a condição de fim-de-arquivo ligando o bit `fail`, que por sua vez é interpretado como `false` no laço. Veja a seguir como ficaria estas mudanças:

```
ifstream ifs( "arquivo.txt" ); // 2 em 1: Abre o "arquivo.txt" E associa ao fluxo 'ifs'.
while( ifs >> val ) {
    if ( val >= 0 and val < 25 ) // Classificar 'val' na primeira faixa.
    ...
}
ifs.close();
```

Infelizmente, esta abordagem, apesar de tecnicamente correta, não é tão robusta se houver problema na formatação dos dados. No exemplo anterior, se houver algum caractere que não seja

parte de um inteiro (uma letra, por exemplo), o laço vai terminar mesmo sem ter lidos todos os dados. Pra priorar, nenhum sinal de erro será gerado. Com isso, o usuário pode ter a falsa impressão de que a leitura e processamento funcionaram corretamente, quando na verdade o programa pode ter falhado em algum momento, se por acaso ele encontrou um dado mal formatado.

Em resumo, o comando `ifs >> val` vai ligar o bit `fail` se um dos dois casos acontecer:

1. Foi alcançado o fim do arquivo, o que significa que não há mais dados para ler, ou;
2. Tentou ler um dado que não pode ser convertido para inteiro.

Uma estratégia para tentar minimizar o problema de um eventual erro de leitura não reportado seria fazer um teste logo após a detecção de um `fail` para saber se foi alcançado o fim do arquivo (final de arquivo correto) ou não (final de arquivo prematuro).

```
if( !ifs.eof() ) {  // Não alcançamos o final do arquivo!
    cerr << "Data error in file!" << endl;
    return EXIT_FAILURE;
}
```

Contudo, pouca informação é fornecida ao usuário, como por exemplo sobre o que causou o erro e onde (no arquivo) ele aconteceu.

Na seção seguinte vamos apresentar uma solução para este problema utilizando leitura de linha do arquivo e o uso de fluxo associado a uma *string*. Mas antes, faça o exercício correspondente, descrito a seguir.

3.4 Exemplo #5: Gravação de dados em arquivo

A gravação de dados em arquivo serve o mesmo esquema apresentado anteriormente para leitura: (1) abrir um fluxo `ofstream` associado a um arquivo; (2) transferir dados para o arquivo de saída orientado a caractere ou linha, ou de maneira formatada, e; (3) fechar o arquivo com `close`.

3.5 Exercício 2: Gravação de dados em arquivo

Neste exercício você deve ler uma sequência de inteiros de comprimento desconhecido a partir de um arquivo de entrada fornecido pelo usuário. Você, então, deve armazenar os números lidos em um `std::vector` e depois abrir um arquivo de saída `invertido.txt` e gravar os mesmo números na ordem inversa que foram lidos, um por linha.

4 Usando Fluxo Associado a String

Considerando que arquivos texto e *strings* são coleções de sequência de caracteres, é de se esperar que sejam tratados de maneira similar. C++ oferece esta possibilidade através da biblioteca `<sstream>`, a qual expõem várias classes que permitem associarmos uma string a um fluxo. Da mesma maneira que usamos o fluxo `ifstream` para ler dados de um arquivo, vamos usar o fluxo `istringstream` para ler dados de uma *string*. Similarmente, a classe `ostringstream` trata o envio de dados para *strings*, da mesma forma que a classe `ofstream` permite gravação de dados em arquivo.

Um exemplo típico de uso de `stringstream` acontece quando queremos criar uma função que converte uma *string* em um inteiro, como por exemplo converter `"1234"` no valor inteiro 1234.

```
/**!
 *  Esta função converte um numeral no formato string em inteiro.
 *  @param inputStr_ Representação de um número inteiro no formato string.
 *  @return Um inteiro correspondente ao número passado como string.
 */
int strToInt( string inputStr_ ) {
    // Criando o fluxo 'iss' e associando-o a 'inputStr_'.
    istringstream iss( inputStr_ );
    int value;
    iss >> value >> std::ws; // Ignora espaço em branco depois.
    if ( iss.fail() or !iss.eof() ) {
        cerr << "strToInt(): Erro, illegal integer format.\n";
    return value;
}
```

Na função `strToInt` a leitura de dados na variável `value` ignora qualquer espaço em branco (ou tabulação) que venha antes (por *default*) dos números ou mesmo depois (com o uso do manipulador `std::ws`). Assim, *strings* como `"␣1234"`, `"1234␣"` ou `"␣1234␣"`, onde '␣' representa espaços em branco, produzem o mesmo valor 1234 em `value`.

Para converter no outro sentido, ou seja, de inteiro para *string* a devemos usar `ostringstream`, como abaixo

```
/**!
 *  Esta função converte um número inteiro em uma string com sua representação.
 *  @param inputInt_ Número inteiro a ser convertido.
 *  @return Retorna uma string correspondente ao número passado como argumento.
 */
string intToStr( int inputInt_ ) {
    // Criando o fluxo 'oss' para receber os dados.
    ostringstream oss;
    oss << inputInt_; // Enviar o inteiro para a string.
    return oss.str(); // Retornar uma string convertida.
}
```

4.1 Exercício 3: Construir `getInteger()`

Normalmente a leitura de um tipo formato a partir do terminal é realizado com `std::cin`. Contudo, você já aprendeu que este tipo de entrada não é robusto o suficiente para lidar com entradas com erros fornecida pelos usuários.

Por exemplo, se tentarmos ler uma variável inteira `val` com `std::cin`:

```
int val;
std::cin >> val;
```

e o usuário digitar, por engano, $12t$ no lugar do valor 126, a variável `val` armazenará o valor incorreto 12.

Desenvolva uma função denominada `int getInteger(string msg_)` que apresenta uma mensagem `msg_` na tela solicitando um inteiro, recebe o inteiro e o retorna para o **código cliente**. A função deve ficar em laço solicitando um valor até que o usuário forneça um valor inteiro correto.

Na sua solução lembre de usar os métodos como `fail`, `eof`, a função `getline` e a nova classe `istringstream`. A estratégia sugerida é usar `getline` com o `std::cin`, associar a *string* lida com um fluxo de *string* para então converter a *string* em inteiro e o retornar para o cliente.

Veja, abaixo, um exemplo de uso da função `getInteger` do ponto de vista do código cliente

```
auto val = getInteger( "Forneca um inteiro: " );
```

4.2 Exercício 4: Tabela Suprema!

No exercício final você deve criar um programa que segue o seguinte roteiro.

1. Usar *argumentos de linha de comando* (i.e. `argc` e `argv`) para receber dois nomes de arquivos de entrada.
2. Ler o primeiro arquivo que contém uma lista de n rótulos, $r_1, r_2, \ldots, r_n$, para compor o cabeçalho de uma tabela, um rótulo por linha, e armazenar em um `vector`;
3. Ler o segundo arquivo de inteiros que deve conter m linhas, cada um com n inteiros e os armazenar em um `vector`. Cada um dos inteiros de uma linha corresponde a um valor associado a um dos rótulos.
4. Produzir uma tabela com cabeçalho no estilo do Exercício 2.3 e imprimir o cabeçalho, simultaneamente, no terminal e em um arquivo de saída de sua escolha.
5. Imprimir, simultaneamente, no terminal e no mesmo arquivo de saída, o corpo da tabela de acordo com os dados lidos, de maneira que os inteiros fiquem alinhados com seus respectivos rótulos.

Veja um exemplo de um arquivo de entrada de rótulos:

```
ITEM
QUANTIDADE
VALOR UNITARIO
```

Abaixo temos um exemplo de um arquivo de entrada de dados da tabela:

```
23 123 985
653 3 5
45 0 1
67 18 15
123 204 10
78 856 234
873 777 7612
```

Por fim, confira um exemplo de um arquivo de saída gerado corretamente, de acordo com as entradas-exemplo:

```
+------+------------+----------------+
| ITEM | QUANTIDADE | VALOR UNITARIO |
+------+------------+----------------+
|   23 |        123 |            985 |
|  653 |          3 |              5 |
|   45 |          0 |              1 |
|   67 |         18 |             15 |
```

```
|   123 |         204 |              10 |
|    78 |         856 |             234 |
|   873 |         777 |            7612 |
+-------+-------------+-----------------+
```

5 Resumo

Este documento apresenta formas básicas de interagir com **fluxo** para extração e inserção de dados. A Orientação a Objetos do C++ permite que os fluxos sejam tratados de forma similar, visto que fazem parte de uma hierarquia de classes.

Em particular, apresentamos fluxos associados ao terminal, arquivo texto e *string*, através de uma breve explicação, exemplos e exercícios. A Tabela 5 apresenta os fluxos estudados e suas fontes de dados.

Direção do fluxo	**Terminal**	**Arquivo texto**	**String**
Entrada	`cin`	`ifstream`	`istringstream`
Saída	`cout`	`ofstream`	`ostringstream`

Tabela 5: Quadro resumo dos fluxos associados a possíveis fontes de dados.

Para cada tipo de fluxo, i.e. de entrada ou de saída, existem um conjunto de métodos e funções que suportam a leitura orientada a caractere ou linha, ou entrada formatada. Cabe a aplicação selecionar o melhor método para efetuar a transferência de informações via fluxo, de acordo com suas necessidades.

◀ FIM ▶

Referências

[1] Eric S. Roberts. *Programming Abstractions in C++*. Pearson Education, Inc. as Addison-Wesley, 1st edition, 2013.