INF 112: Programação II

Aula 17

- ***Streams* de arquivo em C++ - Parte 2**

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

Streams de arquivo em C++ - Parte 2

- Já vimos o uso de `>>`, `<<` e `getline` para ler/escrever dados de um arquivo texto.

- Pode-se ler e escrever dados caractere a caractere também, utilizando-se os métodos `get` e `put`, respectivamente. O programa a seguir ilustra o uso dos mesmos.

```
main () {
  char ch;
  ofstream outfile ("teste.txt");
  do {
    ch=cin.get();
    outfile.put(ch);
  } while (ch!='.');
  outfile.close();
}
```

Arquivo em C++ - Parte 2 – Arquivos binários

➔ Até agora vimos operações para leitura e escrita de dados *byte* a *byte* (*char* a *char*). Mesmo outros tipos, como *int*, são convertidos e gravados caractere por caractere, ocupando um ou mais caracteres (*bytes*) no arquivo.

➔ É possível gravar os dados da forma como estão codificados na memória. Assim, um valor do tipo *int* será sempre escrito com 4 *bytes* (se o sistema codifica *int* com 4 *bytes*) independente de seu valor.

➔ Deste fato, começa uma distinção um tanto confusa entre o que é um *arquivo texto* e o que é um *arquivo binário*. Confusa pelo fato de que, se procurarmos a definição, veremos que muitos distinguem uma coisa e outra pela representação de fim de linha. Nos arquivos texto esta representação é feita com dois caracteres (CR/LF: *carriage return* e *line feed*), enquanto que em arquivos binários somente um caractere é utilizado para o fim de linha.

Continua ...

Arquivo em C++ - Parte 2 – Arquivos binários

➔ Mas veja que isto vale para o Windows. No Linux, por exemplo, somente um caractere é utilizado para o fim de linha sempre, ou seja, por esta simples definição, não haveria distinção entre arquivos texto e binários no Linux.

➔ Ainda nesta linha de definição confusa, há como estipular o modo de abertura de um arquivo como texto (a maneira *default* que vimos até agora) ou binário (veremos a seguir). No Windows é importante usar um ou outro, devido a esta diferença de representação de fim de linha. Já no Linux, isto não é fundamental, pois abrindo de um ou outro modo, pode-se escrever os dados *byte* a *byte* ou na forma como estão representados na memória. Então, não é o tipo de abertura que definirá o formato que será utilizado no arquivo, mas sim os comandos para escrita e leitura.

Continua...

Arquivo em C++ - Parte 2 – Arquivos binários

➔ Outra questão que corrobora para a confusão da distinção é o fato de que qualquer arquivo, mesmo os arquivos tratados como texto são gravados em disco na forma binária, como tudo em computação. O que diferencia a forma de tratamento do arquivo é só a maneira como os dados são divididos (se divide os dados em caracteres – 1 *byte* para cada – ou da forma como são representados na memória volátil).

➔ **Vamos então distinguir aqui arquivos texto de arquivos binários simplesmente pela formatação dos dados. Arquivos texto são aqueles em que a gravação/leitura é feita *byte* a *byte*, enquanto que os binários são aqueles em que a gravação/leitura é feita com a mesma representação em memória RAM.**

Continua...

Arquivo em C++ - Parte 2 – Arquivos binários

- Então:
  - Tratamento *byte* a *byte* (que é o que vimos até o momento) = abertura de arquivo no modo texto.
  - Tratamento dos dados da mesma maneira que são representados em memória primária = abertura de arquivo no modo “binário”.

Arquivo em C++ - Parte 2 – Arquivos binários

- O programa a seguir ilustra esta diferença. Gravamos o número inteiro `123456789` em um arquivo texto e também em um arquivo binário. Após executar o programa, o tamanho do arquivo `teste.txt` será de 9 *bytes*, enquanto que o tamanho do arquivo `teste.bin` será de 4 *bytes* (faça o teste).

- Duas observações. Primeiro, abrimos o segundo arquivo como binário (*flag* `ios::binary`). Segundo, para gravar o inteiro no arquivo binário, do mesmo modo que este dado é representado na memória, utilizamos o método *write*. O método *write* grava um arranjo de caracteres (*bytes*). Como parâmetros devem ser passados o início do arranjo e seu tamanho. No programa a seguir, é passado o endereço da variável *x* (que é convertido para ponteiro de *char* e assim será interpretado como um arranjo de *char*) e o tamanho em *bytes* dessa variável. Note que a função `sizeof` retorna a quantidade em *bytes* que se utiliza para representar um determinado tipo na memória.

Arquivo em C++ - Parte 2 – Arquivos binários

➔ Veja que utilizamos o construtor para abrir o arquivo. Portanto, pode ser assim ou com o uso do método `open`. A vantagem deste último é que controlamos o exato momento em que queremos abrir o arquivo.

```
main() {
  ofstream out1("teste.txt");
  ofstream out2("teste.bin", ios::binary);

  int x = 123456789;

  out1 << x;
  out2.write((char*)&x,sizeof(int));

  out1.close();
  out2.close();
}
```

➔ Experimente abrir o arquivo `teste.bin` no bloco de notas ou no *kate* depois de executar o programa acima. Tente achar uma explicação para o que vê.

Arquivo em C++ - Parte 2 – Arquivos binários

➔ Para se fazer a operação inversa ao `write`, ou seja, ler os dados de um arquivo binário, utiliza-se o método `read`, que tem os mesmos parâmetros do `write`. Vejamos um exemplo mais interessante para gravação e leitura de objetos de uma classe para representar livros.

```
class Livro {
  private:
    char titulo[50];
    char autor[50];
    int numReg;
    float preco;
  public:
    void alteraDados();
    void imprime();
};
...
```

Arquivo em C++ - Parte 2 – Arquivos binários

```
...
void Livro::alteraDados() {
  cout << "\nDigite Titulo : ";
  gets(titulo);
  cout << "\nDigite Autor : ";
  gets(autor);
  cout << "\nDigite o Numero do Registro : ";
  cin >> numReg;
  cout << "\nDigite o Preco : ";
  cin >> preco;
}

void Livro::imprime() {
  cout << "\nTitulo : " << titulo;
  cout << "\nAutor : " << autor;
  cout << "\nNumero do Registro : " << numReg;
  cout << "\nPreco : " << preco;
}
...
```

Arquivo em C++ - Parte 2 – Arquivos binários

- Considere o seguinte programa para gravar livros no arquivo:

```
...
main() {
  ofstream arqLivros;
  Livro li;
  char op;
  arqLivros.open("listaLivros.dat", ios::binary);
  if ( arqLivros.fail() ) {
    cerr << "Problemas ao tentar abrir arquivo!\n";
    return 1;
  }
  do {
    li.alteraDados(); // lê do teclado
    arqLivros.write((char*)&li, sizeof(Livro)); // grava
    cout << "\nInserir outro livro (s/n) ";
    cin >> op;
    cin.ignore(); // Eliminar fim de linha do input buffer
  } while(op == 's');
  arqLivros.close();
}
```

Arquivo em C++ - Parte 2 – Arquivos binários

- Considere o seguinte programa para ler os livros do arquivo:

```
...
main() {
  ifstream arqLivros;
  Livro li;
  arqLivros.open("listaLivros.dat", ios::binary);
  if ( arqLivros.fail() ) {
    cerr << "Problemas ao tentar abrir arquivo!\n";
    return 1;
  }
  while( !arqLivros.eof() ) {
    arqLivros.read( (char*)&li,sizeof(Livro) ); // lê
    if ( !arqLivros.eof() ) { //Se o read não leu eof
      li.imprime(); // imprime no video
      cout << endl;
    }
  }
  arqLivros.close();
}
```

Arquivo em C++ - Parte 2 – Acesso aleatório

➔ Até agora só mostramos como se faz *acesso sequencial* aos dados de um arquivo. A operação que realizamos sempre surte efeito a partir da posição de leitura (chamado de ponteiro *get*) atual ou posição de escrita (chamado de ponteiro *put*) atual do *stream*.

➔ Mas pode ser que queiramos fazer *acesso aleatório* aos dados do arquivo, ou seja, que seja necessário alterar o ponteiro *put*, quando gravando dados, e o ponteiro *get*, quando lendo dados. Para isto, há os seguintes métodos disponíveis:

Função	Descrição
seekg	Movimenta a posição atual de leitura (get)
seekp	Movimenta a posição atual de gravação (put)
tellg	Retorna a posição atual de leitura (em bytes), a partir do início do arquivo
tellp	Retorna a posição atual de gravação (em bytes), a partir do início do arquivo

Arquivo em C++ - Parte 2 – Acesso aleatório

- Os métodos `seekg` e `seekp` têm dois parâmetros. O primeiro diz a quantidade em *bytes* (número positivo ou negativo) de quanto se deslocará o ponteiro. O segundo parâmetro indica o ponto de partida do deslocamento. Há três possibilidades neste caso:
  - `ios::beg` a partir do início do arquivo
  - `ios::cur` a partir da posição corrente (atual)
  - `ios::end` a partir do fim do arquivo

- Assim, suponha que `entrada` seja um objeto `ifstream`. Os seguintes comandos seriam válidos (tudo abaixo vale para `seekp` também):

```
entrada.seekg(100,ios::beg); // Poe no 101º byte do arquivo
entrada.seekg(100); // O mesmo acima (ios::beg é padrão)
entrada.seekg(10, ios::cur); // Anda10 bytes para a frente
entrada.seekg(-10, ios::cur); // Anda 10 bytes para trás
entrada.seekg(-50, ios::end);// 50 para trás a partir do fim
entrada.seekg(0, ios::end); // Posiciona no fim do arquivo
```

Arquivo em C++ - Parte 2 – Acesso aleatório

➔ Vejamos um exemplo. No programa a seguir, utilizamos a classe `Livro`, vista anteriormente. Dado um arquivo já gravado com uma lista de livros, o programa calculará o número de registros gravados e posicionará a leitura no registro desejado pelo usuário.

Arquivo em C++ - Parte 2 – Acesso aleatório

```
...
main() {
  ifstream arqLivros; // cria objeto ifstream
  Livro li; // cria objeto Livro
  arqLivros.open("listaLivros.dat", ios::binary); // abre
  if ( arqLivros.fail() ) {
    cerr << "Problemas ao tentar abrir arquivo!\n";
    return 1;
  }
  arqLivros.seekg(0,ios::end);//coloca ponteiro get no fim
  // Calcula numero de registros:
  long nrec = arqLivros.tellg() / sizeof(Livro);
  cout << "\nNumero de registros : " << nrec;
  cout << "\n\nInsira o numero do registro a exibir ";
  cin >> nrec;
  int posicao = (nrec-1)*sizeof(Livro); // calcula posicao
  arqLivros.seekg(posicao); // posiciona no registro
  arqLivros.read( (char*)&li, sizeof(Livro) ); // lê
  li.imprime(); // imprime no vídeo
  arqLivros.close(); }
```

Arquivo em C++ - Parte 2

➔ Recapitulando e mostrando algumas novidades, têm-se os seguintes modos válidos de abertura de *streams*:

Modos de abertura	Descrição
ios::in	Abre para leitura (default de ifstream).
ios::out	Abre para gravação (default de ofstream),
ios::ate	Abre e posiciona no final do arquivo. (Este modo trabalha com leitura e gravação)
ios::app	Grava a partir do fim do arquivo
ios::trunc	Abre e apaga todo o conteúdo do arquivo
ios::nocreate	Erro de abertura se o arquivo não existe
ios::noreplace	Erro de abertura se o arquivo existir
ios::binary	Abre em binário (default é texto)

Arquivo em C++ - Parte 2

- Quanto a métodos úteis para manipulação de *streams*, além dos que já vimos, vale reforçar o `fail` e mostrar alguns outros referentes a controle de erro.

- Por razões diversas as operações de E/S podem falhar. O programador pode testar o resultado das operações de E/S por meio dos métodos `rdstate`, `good`, `eof`, `fail`, `bad`.

Arquivo em C++ - Parte 2

- Abaixo uma breve descrição dos métodos:

Método	Descrição
rdstate()	Retorna o *flag* que indica erro interno do *stream*.
good()	Retorna *true* se não há qualquer erro e o *stream* está pronto para ser usado.
eof()	Retorna *true* se o *eofbit* estiver ligado devido a tentativa de leitura após fim de arquivo.
fail()	Retorna *true* se o *failbit* estiver ligado devido a entrada inválida ou se *badbit* estiver ligado.
bad()	Retorna true se o *badbit* estiver ligado devido a erro severo de I/O.
clear()	Coloca todos os bits de erro em *off*.

- O exemplo a seguir mostra o uso destes métodos para uma operação de leitura.

Arquivo em C++ - Parte 2

➔ O exemplo a seguir, mostra o uso destes métodos para uma operação de leitura.

Arquivo em C++ - Parte 2

```
main() {
  int x;
  cout << "Antes de uma entrada com falhas:"
    << "\ncin.rdstate(): " << cin.rdstate()
    << "\n cin.eof(): " << cin.eof()
    << "\n cin.fail(): " << cin.fail()
    << "\n cin.bad(): " << cin.bad()
    << "\n cin.good(): " << cin.good()
    << "\n\nEspera inteiro mas entre com um caractere: ";
  cin >> x;
  cout << "\nApos uma entrada com falhas:"
    << "\ncin.rdstate(): " << cin.rdstate()
    << "\n cin.eof(): " << cin.eof()
    << "\n cin.fail(): " << cin.fail()
    << "\n cin.bad(): " << cin.bad()
    << "\n cin.good(): " << cin.good() << "\n\n";
  cin.clear(); // Retorna a estado sem erros
  cout << "Apos cin.clear()"
    << "\ncin.fail(): " << cin.fail()
    << "\ncin.good():" << cin.good()
    << endl << endl; }
```

Arquivo em C++ - Parte 2

➔ A saída para o programa anterior, se entramos com o caractere *i*, será a seguinte:

```
Antes de uma entrada com falhas:
cin.rdstate(): 0
 cin.eof(): 0
 cin.fail(): 0
 cin.bad(): 0
 cin.good(): 1

Espera inteiro mas entre com um caractere: i

Apos uma entrada com falhas:
cin.rdstate(): 4
 cin.eof(): 0
 cin.fail(): 1
 cin.bad(): 0
 cin.good(): 0

Apos cin.clear()
cin.fail(): 0
cin.good():1
```

iomanip

➔ Para finalizarmos a aula de hoje, falemos de uma nova biblioteca relacionada a *streams*: `iomanip`.

➔ Esta biblioteca possui funções para manipulação do fluxo de *streams* com o objetivo de formatação. Com os manipuladores é possível alterar a base de representação dos números inteiros para decimal (dec), octagonal (oct) e hexadecimal (hex). É possível também alterar a precisão de números de ponto flutuante, assim como determinar a tamanho da impressão de um campo.

➔ Veja o exemplo a seguir que lê várias notas de alunos de um arquivo e calcula a média geral. O valor é mostrado de maneira formatada. Experimente executar o programa como está e depois retirando-se toda a formatação, utilizando o `cout` de maneira tradicional. Não esqueça de incluir a biblioteca `iomanip`.

iomanip

```
main() {
  int numNotas;
  float nota, soma, media;
  ifstream arqNotas;
  cout << "Calculo da media de notas em um arquivo:\n\n";
  arqNotas.open("notas.txt");
  if ( arqNotas.fail() ) {
     cerr << "A abertura do arquivo falhou!\n";
     return 1;
  }
  numNotas = 0;
  soma = 0;
...
```

iomanip

```
...
  while ( !arqNotas.eof() ) { // le enquando houver dados
    arqNotas >> nota;         // le proximo valor
    numNotas = numNotas + 1;  // incrementa contador
    soma = soma + nota;       // somatorio das notas
  }
  arqNotas.close();
  cout << "==> " << numNotas << " notas lidas" << endl;
  if ( numNotas > 0 ) {
    media = soma / numNotas;
    // Controlando o formato da saída:
    cout.setf( ios::fixed | ios::showpoint );
    cout.precision(1);
    cout << "Media das notas: " << setw(5) << media;
  } // setw indica quantas colunas a saida irá ocupar
}
```

iomanip

➔ Como vimos, o controle do formato da impressão (e também da leitura) pode ser executado por meio dos manipuladores e *flags* da classe *ios*. São eles:

Nome	Descrição
skipws	Pula brancos na entrada.
left	Ajusta a saída à esquerda.
right	Ajusta a saída à direita.
internal	Adiciona o caracter de preenchimento do campo após o sinal de menos ou indicação de base, mas antes do valor.
dec	Formato decimal para números.
oct	Formato octal para números.
hex	Formato hexadecimal para números.
showbase	Imprime a base indicada (0x para hex e 0 para octal).
showpoint	Força ponto decimal em valores float e preenche com zeros à direita para completar número de casas decimais.
uppercase	Imprime maiúsculo de A a F para saída hexadecimal e E para notação científica.
showpos	Imprime '+' para inteiros positivos.
scientific	Imprime notação científica para float
fixed	Imprime ponto decimal em valores float