UFV
Universidade Federal de Viçosa

INF 112 - Programação II

Arquivos

Introdução

- Em C++, as operações de entrada e saída são realizadas utilizando streams (fluxos).
- Nas operações de entrada, os bytes fluem de um dispositivo para a memória.
- Nas de saída, ocorre o contrário.
- Exemplos de dispositivos:
  - Monitor, teclado, impressora, disco rígido, conexão de rede, etc.

Introdução

- Para utilizar um stream, ele precisa ser, primeiro, aberto (para que ele seja associado ao dispositivo de E/S).
- Após ser utilizado, o stream deverá ser fechado (para liberar o dispositivo de E/S, esvaziar o buffer, etc...)

Introdução

- A biblioteca *iostream* fornece vários recursos para se trabalhar com *streams*.
- Alguns objetos *streams* são criados automaticamente pela biblioteca (e não precisam ser abertos pelo programador):
  - *cin*: *stream* associado à entrada padrão.
  - *cout*: *stream* associado à saída padrão.
  - *cerr*: *stream* de saída normalmente utilizado para enviar mensagens de erro.

Introdução

- Em geral, a entrada padrão é lida a partir do teclado e a saída padrão é enviada para a tela (mas isso pode ser modificado pelo usuário).
- O *cout* utiliza saída *bufferizada*, enquanto o *cerr* não utiliza buffer.

Introdução

- Ao trabalhar com um stream (por exemplo, o cout), normalmente o programador utiliza um operador para inserir/extrair dados do fluxo.
- Ao utilizar o comando "cout << teste;", o operador de inserção de fluxo é utilizado para enviar os dados da variável teste (que está na memória) para o dispositivo de saída associado ao cout.

Arquivos

- A biblioteca fstream fornece classes cujos objetos podem ser utilizados para manipular arquivos.
- As principais classes utilizadas para se criar arquivos lógicos são:
  - **ifstream**: objetos do tipo ifstream são streams de entrada (leitura de arquivos).
  - **ofstream**: objetos do tipo ofstream são streams de saída (escrita em arquivos).
  - **fstream**: objetos do tipo fstream podem ser utilizados para se realizar entrada e saída em um arquivo.

Arquivos

- Para ilustrar o uso de arquivos em C++, vamos considerar o seguinte problema:
- Vamos desenvolver um programa que lê as notas (de 0 a 100) de alguns alunos do teclado e, então, imprima na tela as notas normalizadas (de forma que a maior nota da sala seja transformada em 100).
- Inicialmente, será informado ao programa o número de notas que serão lidas.
- O programa, inicialmente, será desenvolvido de modo a ler os dados de entrada a partir do *cin* e gravar a saída no *cout*.

```
...
    float *notas;
    int n;
    cin >> n;
    notas = new float[n];
    for(int i=0;i<n;i++)
        cin >> notas[i];

    float mx = 0;
    for(int i = 0; i < n; i++)
        mx = max(mx, notas[i]);

    for(int i = 0; i < n; i++)
        cout << 100 * notas[i] / mx << endl;

    delete []notas;
...
```

Arquivos

- Vamos, agora, adaptar o programa de modo que ele leia os dados de entrada do arquivo "notas.txt" e escreva a saída no arquivo "notasNormalizadas.txt".
- Vamos supor que os dados de entrada/saída estarão no mesmo formato utilizado no exemplo anterior.
- Notem que no exemplo anterior os dados foram lidos/gravados nos *streams* (padrões) *cin* e *cout*.
- Para podermos utilizar a leitura/gravação em arquivos, vamos ter que criar nossos próprios streams.

```
...
#include <fstream>
...
    ifstream fin("notas.txt");
    ofstream fout("notasNormalizadas.txt");
    fin >> n;
    notas = new float[n];
    for(int i = 0; i < n; i++)
        fin >> notas[i];

    float mx = 0;
    for(int i = 0; i < n; i++)
        mx = max(mx, notas[i]);

    for(int i = 0; i < n; i++)
        fout << 100 * notas[i] / mx << endl;

    fin.close();
    fout.close();
    delete []notas;
...
```

Arquivos

- Os *streams* apresentados no exemplo anterior também podem ser construídos utilizando um construtor default. Nesse caso, para ser utilizado eles deverão ser, primeiro, abertos utilizando o método *open*.

```
ifstream fin("notas.txt");
ofstream fout("notasNormalizadas.txt");
```

```
ifstream fin;
ofstream fout;
fin.open("notas.txt");
fout.open("notasNormalizadas.txt");
```

Arquivos

- Principais operadores e funções-membro de *file streams*.

Função	Uso	Exemplo
open	Abre um arquivo	fin.open(“teste.txt”);
close	Fecha um arquivo	fin.close();
>>	Extrai dados de um stream de entrada	fin >> x;
<<	Insere dados em um stream de saída	fout << “teste” << endl;
get	Lê um caractere de um stream (inclusive espaços)	char c = fin.get(); If (c==EOF) … //EOF = caractere de fim de arquivo .. normalmente Ctrl+d no Unix e Ctrl+z no Windows.
eof	Retorna true após o usuário tentar ler dados após o último caractere do stream de entrada	if (fin.eof()) ...
fail	Retorna true se ocorrer algum erro no formato dos dados sendo lidos (ex: tentar ler um inteiro mas o fluxo fornecer um caractere).	int x; fin > x; if (fin.fail()) cerr << “Erro” <<endl;

Arquivos

- Exercício: modifique o exemplo apresentado (de normalização de notas) de forma que ele leia um arquivo de notas onde o primeiro número não representa a quantidade de notas (ou seja, seu programa deverá descobrir a quantidade de notas no arquivo com base no fim de arquivo (EOF)).
- Obs: quando o *stream* de entrada é lido a partir do teclado, o fim de arquivo pode ser enviado para o *stream* digitando o caractere Ctrl+z (Windows) ou Ctrl+d (Linux/Unix/Mac).

```
...
    ifstream fin("notas2.txt");
    ofstream fout("notasNormalizadas2.txt");
    int n = 0;
    float *notas = new float[1000];
    while(true) {
        fin >> notas[n];
        if (fin.eof())
            break;
        else
            n++;
    }

    float mx = 0;
    for(int i=0;i<n;i++)
        mx = max(mx, notas[i]);
    for(int i=0;i<n;i++)
        fout << 100*notas[i]/mx << endl;
    fin.close();
    fout.close();
    delete []notas;
...
```

Arquivos

- Quando um arquivo é aberto para leitura, é possível verificar se a operação de abertura ocorreu de forma correta utilizando o método *is_open* (a operação pode falhar no caso do arquivo não existir, do arquivo estar bloqueado, etc.)
- O mesmo pode ser feito para o caso de arquivos de saída (a criação do arquivo pode falhar).

Arquivos

- Por padrão, quando um arquivo de saída é aberto, se ele não existe ele é criado e, se ele já existir, seu conteúdo original é apagado.
- A flag "ios::app" pode ser utilizada durante a abertura de um arquivo de saída para que os dados gravados no arquivo sejam concatenados (ou seja, o conteúdo original não será apagado). Veja o slide seguinte.

Arquivos

- Principais flags que podemos utilizar durante a abertura de um arquivo

Flag	Uso	Exemplo
ios::app	Configura a abertura do arquivo de modo que os dados gravados sejam concatenados no final do arquivo.	fin.open(“teste.txt”, ios::app);
ios::in	Abre o arquivo no modo de leitura	fin.open(“teste.txt”, ios::in);
ios::out	Abre o arquivo no modo de escrita (padrão para arquivos do tipo ofstream, opicional para fstream).	fin.open(“teste.txt”, ios::out);

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

using namespace std;

int main() {
    ofstream saida("teste.txt");
    saida << "abc" << endl;
    saida.close();
    return 0;
}
```

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

using namespace std;

int main() {
    ofstream saida("teste.txt”, ios::app);
    saida << "abc" << endl;
    saida.close();
    return 0;
}
```

Arquivos

- As classes *ifstream*/*ofstream* oferecem vários métodos para manipulação de arquivos.
  - O método *get* da classe *ifstream* lê um caractere (inclusive espaço em branco) de um arquivo de entrada e o retorna.
  - O método *put* da classe *ofstream* recebe como argumento um caractere e o escreve no arquivo de saída.
  - O método *ignore* da classe *ifstream* lê caracteres do *stream* de entrada e os ignora. O *ignore* pode receber dois argumentos: o primeiro argumento (que, por padrão, vale 1, indica o número de caracteres que serão ignorados e o segundo argumento indica um delimitador (que, por padrão, vale EOF). O *ignore*, ao ser executado, lê caracteres e os descarta até que a quantidade de caracteres informada seja descartada ou até o delimitador ser atingido.

```
...
ifstream entrada("teste.txt");
while(true) {
   char c = entrada.get();
   if (entrada.eof())
         break;
      cout << c << endl;
      entrada.ignore(10000, ' ');
}
...
```

Arquivos

- Exercício: faça um programa que recebe como argumento o nome de dois arquivos e, então, copia o texto do primeiro arquivo para o segundo.

```
int main(int argc, char **argv) {
    if (argc!=3) {
        cerr << "Erro, use: copia Origem Destino" << endl;
        exit(1);
    }
    ifstream entrada(argv[1]);
    ofstream saida(argv[2]);
    if (!entrada.is_open()) {
        cerr << "Erro ao abrir arquivo de entrada" << endl;
        exit(1);
    }
    if (!saida.is_open()) {
        cerr << "Erro ao abrir arquivo de saida" << endl;
        exit(1);
    }
    char c;
    while(true) {
        c = entrada.get();
        if (c==EOF)
            break;
        saida.put(c);
    }

    entrada.close();
    saida.close();
    return 0;
}
```

Arquivos

- Outro método bastante importante da classe ifstream é o método getline.

  istream& getline (char s, streamsize n);*

- Quando chamado, o método getline extrai até (n-1) caracteres do stream e os armazena em s. Se chegar no delimitador de linhas ('\n'), a leitura é interrompida e o '\n' é descartado (ele é lido do stream mas não é gravado em s).

```
int main(int argc, char **argv) {
    if (argc!=3) {
        cerr << "Erro, use: teste Origem Destino" << endl;
        exit(1);
    }
    ifstream entrada(argv[1]);
    ofstream saida(argv[2]);
    if (!entrada.is_open()) {
        cerr << "Erro ao abrir arquivo de entrada" << endl;
        exit(1);
    }
    if (!saida.is_open()) {
        cerr << "Erro ao abrir arquivo de saida" << endl;
        exit(1);
    }
    char temp[1000];
    int i =1;
    while(true) {
        entrada.getline(temp,1000);
        if (entrada.eof()) break;
        saida << "Linha " << i++ << " : " << temp << endl;

    }
    entrada.close();
    saida.close();
    return 0;
}
```

Arquivos

- Por enquanto, os exemplos que vimos armazenam e lêem texto em arquivos.
- Dessa forma, o trecho de código abaixo gera um arquivo onde o conteúdo é o "texto" 1234.

```
int x = 1234567;
fout << x << endl;
```

```
$./a.out
$cat saida.txt
1234567
```

Arquivos

- Em geral, os números do tipo "int" utilizam 4 bytes de memória.
- Se gravarmos um arquivo de texto com 1000 números, quantos bytes utilizaremos?

Arquivos

- Em geral, os números do tipo "int" utilizam 4 bytes de memória.
- Se gravarmos um arquivo de texto com 1000 números, quantos bytes utilizaremos?
  - Isso depende do número de dígitos nos números que serão gravados!
  - Além disso, precisaremos utilizar um caractere para separar os números (ex: espaço em branco).

Arquivos

- É possível gravar/ler dados em arquivos de forma que eles fiquem armazenados de maneira similar à que eles ficam na memória (ou seja, um inteiro ocuparia 4 bytes independente do número de caracteres utilizados). Tais arquivos são chamados de arquivos "binários" (os exemplos vistos nas últimas aulas são de arquivos de "texto").
- Para isso, basta abrir o arquivo passando a flag "ios::binary" e utilizar os métodos de gravação/leitura de dados não formatados (ex: *read* e *write*).

Arquivos

- Na verdade, os arquivos binários e arquivos de texto são armazenados de forma semelhante no disco.
- A diferença entre eles é mais conceitual e ocorre no software que os utiliza. Por exemplo, ao ler o byte 01000001 o programa pode interpretar tal byte como o caractere com código 65 (o caractere 'A') ou então como parte dos 4 bytes que formam um número inteiro .
- O exemplo a seguir ilustra a gravação de dados em formato binário.

```cpp
int main() {
    ofstream saida("teste1.txt");
    ofstream saida2("teste1.bin",ios::binary);

    int num = 123456;
    saida << num;
    saida2.write(reinterpret_cast<char *>(&num), sizeof(num));


    saida.close();
    saida2.close();
    return 0;
}
```

```
$./a.out
$ls -la
....
-rw-r--r-- 1 gvc gvc     4 Dec  5 14:38
teste1.bin
-rw-r--r-- 1 gvc gvc     6 Dec  5 14:38
teste1.txt
....
$cat teste1.bin
@@
$cat teste1.txt
123456
```

Arquivos

- O comando *cat* do linux exibe o conteúdo de um arquivo o interpretando em modo "texto".
- Note que o arquivo binário apresenta alguns caracteres "estranhos" (ele contém 4 caracteres, mas 2 deles são caracteres "não imprimíveis").
- Veja um outro exemplo:

```cpp
int main() {
    ofstream saida("teste1.txt");
    ofstream saida2("teste1.bin",ios::binary);

    int num = 1094795585;
    saida << num;
    saida2.write(reinterpret_cast<char *>(&num),sizeof(num));


    saida.close();
    saida2.close();
    return 0;
}
```

```
$./a.out
$ls -la
....
-rw-r--r-- 1 gvc gvc     4 Dec  5 14:38
teste1.bin
-rw-r--r-- 1 gvc gvc     10 Dec  5 14:38
teste1.txt
....
$cat teste1.bin
AAAA
$cat teste1.txt
1094795585
```

Arquivos

- Por que o arquivo teste1.bin (que é binário) contém um texto (“AAAA”)?

Arquivos

- Por que o arquivo teste1.bin (que é binário) contém um texto ("AAAA")?
  - O número 1094795585 é representado, em binário, pela sequencia de bytes abaixo:
    01000001 01000001 01000001 01000001
  - O cat interpreta cada byte como um caractere.
  - O número 01000001 representa o número 65 em decimal.
  - 65 é o código ASCII para o caractere A!
- Este exemplo reforça a ideia de que arquivos binários e de texto são basicamente a mesma coisa! A diferença está na forma com que os programas os interpretam.

Arquivos

- Funcionamento do método *write*:
  - O método write recebe como argumentos um apontador para char (esse apontador deve apontar para o primeiro byte dos dados que serão gravados) e o número de bytes que serão gravados no stream.
  - No exemplo que mostramos, o número de bytes a serem gravados no stream é *sizeof(int)*.
  - Se o usuário desejar gravar, por exemplo, um array de inteiros em um stream, esse array pode ser gravado com a execução de um único comando *write*.
  - Exercício: como ficaria o código para gravar um array dessa forma?

```
int main() {
    ofstream saida("teste1.bin",ios::binary);

    int n = 10;
    int v[n];
    for(int i=0;i<n;i++) v[i] =i;
    saida.write(reinterpret_cast<char *>(&n),sizeof(int));
    saida.write(reinterpret_cast<char *>(v),sizeof(int)*n);


    saida.close();
    return 0;
}
```

Neste exemplo, antes de gravar o array gravamos também o número de elementos no array (para facilitar a identificação do tamanho do array quando algum programa for abrir o arquivo).

Por que passamos "v" (e não "&v") como argumento do *write*?

Arquivos

- Funcionamento do método *read*:
  - O método *read* recebe como argumentos um apontador para char (esse apontador deve apontar para o primeiro byte de memória onde os dados lidos do stream serão gravados) e o número de bytes que serão lidos do stream.
  - Exercício: faça um programa para ler o array gravado no exemplo anterior e imprimí-lo na tela.

```cpp
int main() {
    ifstream entrada("teste1.bin",ios::binary);

    int n;
    entrada.read(reinterpret_cast<char *>(&n),sizeof(int));
    int v[n];
    entrada.read(reinterpret_cast<char *>(v),sizeof(int)*n);

    for(int i=0;i<n;i++) cout << v[i] << endl;

    entrada.close();
    return 0;
}
```

Arquivos

- Os métodos *read* e *write* podem ser utilizados facilmente para *ler/gravar* objetos de classes (ou instâncias de *structs*) criadas pelo programador.
- Veja o exemplo seguinte:

```
class Aluno {
    public:
        char nome[50];
        int matricula;
};

int main() {
    ofstream saida("teste1.bin",ios::binary);

    int n = 2;
    Aluno v[n];

    strcpy(v[0].nome, "salles viana");
    v[0].matricula = 123;

    strcpy(v[1].nome, "qwer dgsdfg");
    v[1].matricula = 5557;

    saida.write(reinterpret_cast<char *>(&n),sizeof(int));
    saida.write(reinterpret_cast<char *>(v),sizeof(Aluno)*n);


    saida.close();
    return 0;
}
```

```
class Aluno {
    public:
        char nome[50];
        int matricula;
};

int main() {
    ifstream entrada("teste1.bin",ios::binary);

    int n;
    entrada.read(reinterpret_cast<char *>(&n),sizeof(int));

    Aluno v[n];
    entrada.read(reinterpret_cast<char *>(v),sizeof(Aluno)*n);

    for(int i=0;i<n;i++) cout << v[i].matricula << " ::: "
                              << v[i].nome << endl;


    entrada.close();
    return 0;
}
```

Arquivos

- Vamos desenvolver, agora, um programa que imprime os caracteres que estão em um intervalo de posições em um arquivo (ex: imprimir todos 501 os caracteres que estão entre o caractere 1000 e o caractere 1500 do texto).
  - O programa deverá funcionar da seguinte forma:
  - O usuário executa o programa passando como argumento para ele um arquivo de texto.
  - O programa então, pede para o usuário digitar a posição inicial e final dos caracteres que serão impressos.
  - Após imprimir os caracteres, o programa repete o processo, pedindo para o usuário digitar outro intervalo.
  - Vamos supor que, por algum motivo, o arquivo não possa ser copiado inteiramente para a memória (ex: o arquivo é muito grande).

```
int main(int argc,char **argv) {
    if (argc!=2) { cerr << "Erro: o programa requer …." << endl;
        Exit(1);    }
    while(true) {
        ifstream fin(argv[1]);
        if (!fin.is_open()) { cerr << "Erro ao tentar abrir o
                                      arquivo" <<endl;
            exit(1);    }
        int begin, end;
        cout << "Digite o intervalo de caracteres que será impresso (
                 -1 -1 para parar): " ;
        cin >> begin >> end;
        if (begin <= 0 || end <=0 ) {
            fin.close();
            break;
        }
        char c;
        for(int i=0;i<begin-1;i++) fin.get(); //le e descarta os primeiros"begin-1" caracteres...
        cout << "--------Conteudo-------" <<endl;
        for(int i=begin;i<=end;i++) {
            c = fin.get();
            cout << c;
        }
        cout << "\n-----------------------" << endl;
        fin.close();
    }
    return 0;
}
```

Arquivos

- Qual o problema do código anterior?

Arquivos

- Qual o problema do código anterior?
  - Se o arquivo for muito grande, o programa poderá ter que ler muitos caracteres desnecessários.
  - Isso poderia deixar o programa lento.
- Esse problema ocorre porque o acesso que estamos fazendo no arquivo é um acesso "sequencial". Ou seja, a leitura começa no início do arquivo e ele é lido sequencialmente até alcançarmos os dados que efetivamente precisamos acessar.

Arquivos

- Ao abrir um arquivo para leitura, o ifstream possui um ponteiro (chamado de "get") que, inicialmente aponta para o primeiro byte do arquivo (os arquivos de escrita possuem um ponteiro chamado "put").
- À medida em que os dados são lidos, esse ponteiro é deslocado para a frente.
- As classes ifstream/ofstream/fstream fornecem métodos para modificar a posição desses ponteiros. Os acessos que utilizam tais métodos são chamados de "acessos aleatórios".

Arquivos

- Métodos para realizar acessos aleatórios em arquivos.

Método	**Descrição**
seekg(int n, ios_base::seekdir dir)	Desloca o ponteiro get em “n” unidades.
seekp(int n, ios_base::seekdir dir)	Desloca o ponteiro put em “n” unidades.
tellg()	Retorna a posição do ponteiro get.
tellp()	Retorna a posição do ponteiro put.

- A flag *dir* indica em relação a que o deslocamento é realizado.

Flag	**Descrição**
ios::beg	Deslocamento em relação ao início do arquivo.
ios:: end	Deslocamento em relação ao final do arquivo.
ios::cur	Deslocamento em relação à posição corrente.

Arquivos

- Exemplos de deslocamentos em arquivos.

```
fin.seekg(10,ios::cur); //Desloca 10 bytes para a frente
fin.seekg(5*sizeof(int),ios::cur); //Desloca para a frente o número de
                posições relativas a 5 inteiros
fin.seekg(-10,ios::cur); //Descloca 10 bytes para a trás

fout.seekp(-10,ios::end); //Desloca 10 bytes para trás em relação ao
                           fim do arquivo.
fout.seekp(10,ios::beg); //Posiciona o ponteiro no decimo primeiro
                          byte do arquivo
fout.seekp(0,ios::beg); //Posiciona o ponteiro no inicio do arquivo

//Posiciona o ponteiro no final do arquivo e, a seguir, pega o tamanho
                                do arquivo
fin.seekg(0,ios::end);
int tamArquivo = fin.tellg();
```

Arquivos

- Obs: as operações de deslocamento em arquivos podem ser realizadas tanto em arquivos de texto quanto em arquivos binários.
- Exercício:
  - Modifique o programa de leitura de caracteres de arquivos de modo que ele utilize deslocamentos para fazer acessos aleatórios no arquivo de entrada.
  - Adicione um tratamento para evitar que o final do intervalo de leitura esteja após o final do arquivo.

```
// ...
    ifstream fin(argv[1]);
    if (!fin.is_open()) { cerr << "Erro ao tentar abrir o arquivo" <<endl;
            exit(1);    }
    fin.seekg(0,ios::end);
    int tamArquivo = fin.tellg(); //descobre o tamanho do arquivo
    cerr << "O arquivo tem: " << tamArquivo << " bytes" <<endl;
    while(true) {
        int begin, end;
        cout << "Digite o intervalo de caracteres que será impresso (-1 -1 para parar): " ;
        cin >> begin >> end;
        if (begin <= 0 || end <=0 ) {
            break;  }

        fin.seekg(begin-1,ios::beg);
        if(end > tamArquivo)
            end = tamArquivo;

        cout << "--------Conteudo-------" <<endl;
        for(int i=begin;i<=end;i++) {
            char c = fin.get();
            cout << c;
        }
        cout << "\n-----------------------" << endl;
    }
    fin.close();
// ...
```