

Aula 8

Manipulação de Arquivos em C++
Programação II

Prof. Sandino Jardim | CC-UFMT-CUA



Definição

- Estruturas para armazenar dados de forma **persistente**
- Tipos:
 - Texto: Dados legíveis por humanos
 - Binário: Dados no formato de máquina
- Aplicações
 - Logs, armazenamento de configurações, banco de dados, etc.



Abertura e Fechamento

- Cabeçalho: `#include <fstream>`
- Classes
 - `ifstream` – Leitura de arquivos
 - `ofstream` – Escrita em arquivos
 - `fstream` – Leitura e escrita
- Modos de abertura:
 - `ios::in`
 - `ios::out`
 - `ios::app`
 - `ios::binary`

```
std::ifstream inputFile("dados.txt");  
if (!inputFile) {  
    std::cerr << "Erro ao abrir o arquivo!";  
    return 1;  
}  
inputFile.close();
```



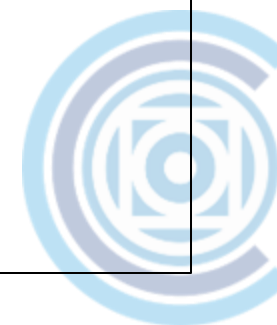
Leitura de Arquivos

- Linha por linha

```
std::string linha;  
std::ifstream inputFile("dados.txt");  
while (std::getline(inputFile, linha)) {  
    std::cout << linha << std::endl;  
}
```

- Palavra por palavra

```
std::string palavra;  
std::ifstream inputFile("dados.txt");  
while (inputFile >> palavra) {  
    std::cout << palavra << std::endl;  
}
```



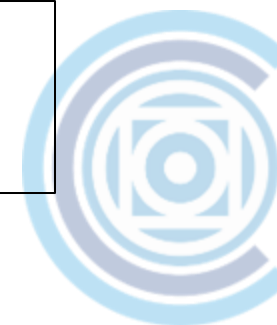
Escrita em Arquivos

- Escrevendo em Arquivo

```
std::ofstream outputFile("saida.txt");  
outputFile << "Primeira linha" << std::endl;  
outputFile << "Segunda linha" << std::endl;  
outputFile.close();
```

- Append (adicionar ao final)

```
std::ofstream outputFile("saida.txt", std::ios::app);  
outputFile << "Nova linha adicionada" << std::endl;  
outputFile.close();
```

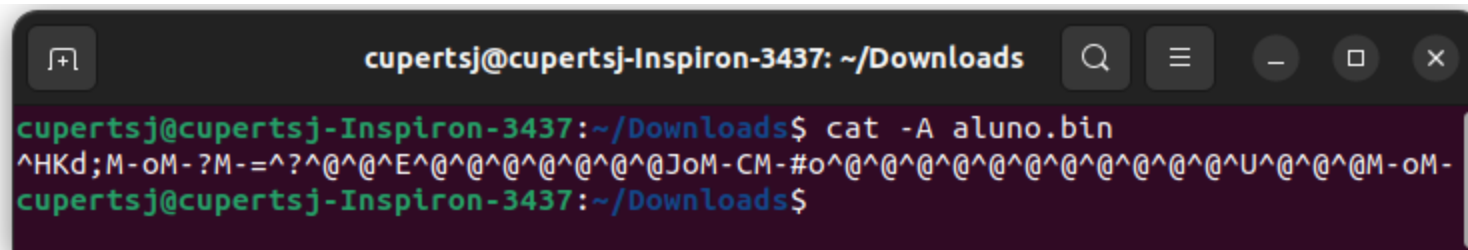


Manipulação de Arquivos Binários

- Vantagens de arquivos binários:
 - Tamanho compacto
 - Velocidade de leitura e escrita
 - Acesso aleatório mais eficiente
 - Fidelidade dos dados
 - Segurança e privacidade
 - Manipulação de dados complexos

```
class Aluno {  
private:  
    string nome;  
    int idade;  
public:  
    Aluno(string nome, int idade) :  
        nome(nome), idade(idade) {}  
};  
  
Aluno j("João", 21);  
std::ofstream file("aluno.bin", std::ios::binary);  
file.write(reinterpret_cast<char*>(&j), sizeof(j));  
file.close();
```

CC-UFMT-CUA



```
cupertsj@cupertsj-Inspiron-3437: ~/Downloads  
cupertsj@cupertsj-Inspiron-3437:~/Downloads$ cat -A aluno.bin  
^HKd;M-oM-?M-=^?^@^@^E^@^@^@^@^@^@JoM-CM-#o^@^@^@^@^@^@^@^@^@U^@^@^@M-oM-  
cupertsj@cupertsj-Inspiron-3437:~/Downloads$
```



Leitura de Arquivos Binários

- Exemplo de Leitura:

```
Aluno a;  
std::ifstream file("aluno.bin", std::ios::binary);  
file.read(reinterpret_cast<char*>(&a), sizeof(a));  
file.close();  
  
std::cout << "Nome: " << a.nome << "\nIdade: " << a.idade << std::endl;
```



Separação Interface/Implementação

- Atua junto da estratégia de separação de interface e implementação em arquivos separados

```
// ---- Sales_data.h ----
// #includes should appear before opening the ns
#include <string>
namespace cplusplus_primer {
    class Sales_data { /* . . . */};
    Sales_data operator+(const Sales_data&, const Sales_data&);
    // declarations for the remaining functions
}

// ---- Sales_data.cc ----
// be sure any #includes appear before opening the ns
#include "Sales_data.h"
namespace cplusplus_primer {
    // definitions for Sales_data members and overloaded operators
}
```

```
// ---- user.cc ----
// names in the Sales_data.h header
// are in the cplusplus_primer ns
#include "Sales_data.h"
int main()
{
    using cplusplus_primer::Sales_data;
    Sales_data trans1, trans2;
    // . . .
    return 0;
}
```



Namespaces – Definição de membros

- Códigos internos a um espaço de nomes podem usar a forma curta dos nomes

```
namespace cplusplus_primer {  
    // reopen cplusplus_primer  
    // members defined inside the namespace may use unqualified names  
    std::istream&  
    operator>>(std::istream& in, Sales_data& s) { /* . . . */}  
}
```

- Também é possível definir um membro de um *ns* fora dele:

```
// namespace members defined outside the namespace must use qualified names  
cplusplus_primer::Sales_data  
cplusplus_primer::operator+(const Sales_data& lhs,  
const Sales_data& rhs)  
{  
    Sales_data ret(lhs);  
    // . . .  
}
```



Namespace – Global e aninhamento

- Implicitamente declarado, pode ser referenciado pelo operador de escopo (sem nome) → `::member_name`
- Podem ser aninhados:

```
namespace cplusplus_primer {  
    // first nested namespace: defines the Query portion of the library  
    namespace QueryLib {  
        class Query { /* . . . */ };  
        Query operator&(const Query&, const Query&);  
        // . . .  
    }  
    // second nested namespace: defines the Sales_data portion of the library  
    namespace Bookstore {  
        class Quote { /* . . . */ };  
        class Disc_quote : public Quote { /* . . . */ };  
        // . . .  
    }  
}
```

`cplusplus_primer::QueryLib::Query`



Namespaces - Aliases

- Permite associar um sinônimo mais curto a um ns definido

```
namespace cplusplus_primer  
{ /* . . . */ };
```

```
namespace primer = cplusplus_primer;
```

```
namespace Qlib = cplusplus_primer::QueryLib;  
Qlib::Query q;
```



A declaração `using`

- Permite apresentar a utilização de um *namespace* por vez
- Nomes introduzidos pela declaração obedecem regras normais de escopo:
 - São visíveis desde a declaração até o final do escopo onde a declaração aparece
 - Entidades com o mesmo nome definido em um escopo externo são escondidas
 - Nomes não qualificados podem ser usados somente dentro do escopo onde houve a declaração
 - Encerrado o escopo, nomes qualificados devem ser utilizados



A declaração `using`

- Exemplo

```
#include <iostream>
#include <string>
using std::string;
int main()
{
    string str = "Example";
    using std::cout;
    cout << str;
}
```



A diretiva `using`

- Diferente da declaração, permite usar a forma não qualificada de todos os nomes definidos em um *namespace*
- Seu uso indiscriminado reintroduz os problemas de colisão inerentes à utilização de múltiplas bibliotecas



A diretiva `using` - escopo

- Torna o espaço de nomes disponível desde o escopo global do programa.

```
// namespace A and function f are defined at global scope
namespace A {
    int i, j;
}
void f()
{
    using namespace A;
    // injects the names from A into the global scope
    cout << i * j << endl; // uses i and j from namespace A
    // . . .
}
```



A diretiva using - escopo

- Outro exemplo:

```
namespace blip {  
    int i = 16, j = 15, k = 23;  
    // ...  
}  
int j = 0; // ok: j inside blip is hidden inside a namespace  
void manip()  
{  
    // using directive; the names in blip are ‘added’ to the global scope  
    using namespace blip;    // clash between ::j and blip::j  
                             // detected only if j is used  
  
    ++i;                    // sets blip::i to 17  
    ++j;                    // error ambiguous: global j or blip::j ?  
    ++::j;                  // ok: sets global j to 1  
    ++blip::j;              // ok: sets blip::j to 16  
    int k = 97;             // local k hides blip::k  
    ++k;                    // sets local k to 98  
}
```



A diretiva `using` - precauções

- Reintroduz riscos à colisão de nomes em programas grandes;
- Torna programas vulneráveis à atualização de bibliotecas que venham a declarar nomes conflitantes;
- Erros de longo prazo podem aparecer à medida que a biblioteca for sendo mais explorada
- Uso da declaração é mais indicado, pois oferece maior controle sobre erros;
- Diretivas são mais úteis nos arquivos de implementação do próprio contexto do *namespace*

