Recursos Avançados de C++

Módulo 3

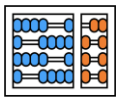
Prof. Dr. Bruno B. P. Cafeo

Instituto de Computação
Universidade Estadual de Campinas

Agenda

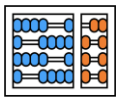
- Apresentação da disciplina

- Recapitulação de conceitos
  - Manipulação de Arquivos e Diretórios
  - Serialização e Persistência de Dados



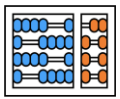
Horário de aulas

Dia da Semana	Horário	Local	Tipo
Segunda-feira	08h00 - 10h00	IC 351	Teórica
Quarta-feira	08h00 - 10h00	IC 351	Teórica



PEDs e PAD

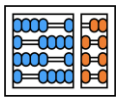
Nome	E-mail
Cristiano Gabriel de Souza Campos	c204189@dac.unicamp.br
Luana Amorim Lima	l197620@dac.unicamp.br



UNICAMP

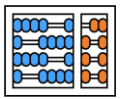
Atendimento

- Terça-feira – 18h00 ~ 19h00 – Bruno
- Quarta-feira – 12h00 ~ 13h00 – Luana
- Quarta-feira – 19h00 ~ 20h00 – Cristiano
- Sexta-feira – 12h00 ~ 13h00 – Luana
- Sexta-feira – 18h00 ~ 19h00 - Cristiano



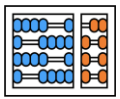
Programa da disciplina

- Manipulação de arquivos e diretórios
- Serialização e persistência de dados
- Gerenciamento de erros e depuração de código
- Boas práticas de programação e padrões para C++
- Diferenças entre as versões de C++
- Tratamento de exceções
- Templates e programação genérica
- Smart pointers e semântica de transferência
- Desenvolvimento de C++ utilizando Win32 API
- Overview de Standard Template Library (STL)
- Active Template Library (ATL)
- Overview de MFC



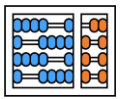
Submissão de atividades

- Os trabalhos práticos e projetos realizados durante a disciplina deverão ser submetidos via Google Classroom.



Página do curso

- O material da disciplina ficará disponível na sala do Google Classroom

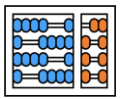


UNICAMP

Avaliações e composição de notas

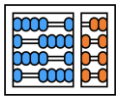
- Exercício prático (P)
  - 5 exercícios (1 já foi entregue)

- Média final (MF)
  - MF = 0,2*E1 + 0,2*E2 + 0,2*E3 + 0,2*E4 + 0,2*E5



Datas*

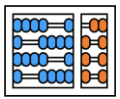
#	Data	Dia da semana	Conteúdo/Atividade	Observação
1	09/10	Segunda-feira	- Apresentação da disciplina - Manipulação de diretórios - Serialização e persistência	
2	11/10	Quarta-feira	- Gerenciamento erros e depuração - Tratamento de exceções	EXERCÍCIO 2
3	16/10	Segunda-feira	- Templates e programação genérica - Smart pointers e semântica de transferência	EXERCÍCIO 3
-	18/10	Quarta-feira	NÃO HAVERÁ AULA	
4	23/10	Segunda-feira	- Desenvolvimento em C++ Utilizando Win32 API	EXERCÍCIO 4
5	25/10	Quarta-feira	- Overview de Standard Template Library (STL)	
6	30/10	Segunda-feira	- Overview de Active Template Library (ATL)	EXERCÍCIO 5



* Sujeito a alterações

UNICAMP

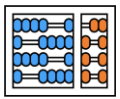
Manipulação de diretórios



UNICAMP

std::filesystem (Biblioteca de Sistema de Arquivos)

- Funcionalidades Principais:
  - Manipulação avançada de arquivos e diretórios.
  - Abertura, leitura e escrita de arquivos.
  - Trabalha com objetos de caminho (std::filesystem::path) e fornece funções para operações de sistema de arquivos.
- Uso Típico:
  - Manipulação de diretórios, navegação e busca de arquivos.
  - Manipulação segura de caminhos de arquivo.
  - Leitura e escrita avançadas de arquivos.
- Diferenças Notáveis:
  - Ideal para operações de sistema de arquivos e diretórios.
  - Fornece funcionalidades robustas para manipular caminhos e diretórios.

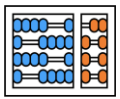


std::filesystem::path

- Representa caminhos de diretórios e arquivos de forma portável.

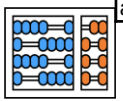
- Métodos para manipular caminhos, como parent_path(), filename(), e outros.

```
fs::path caminho("C:\\pasta\\arquivo.txt");
```



std::filesystem::path

Construtor	Propriedade Prática
path() noexcept;	Cria um objeto std::filesystem::path vazio, que é útil quando você precisa de um objeto de caminho inicialmente sem valor definido. Por exemplo, você pode usá-lo como um ponto de partida para construir caminhos incrementalmente ao adicionar diretórios e nomes de arquivos posteriormente.
path(const path& p);	Permite fazer uma cópia exata de um caminho existente. Isso é útil quando você deseja duplicar um caminho sem modificar o original. Por exemplo, se você estiver gerenciando uma lista de caminhos de arquivos e desejar criar uma cópia de um caminho para fins de manipulação separada, pode usar esse construtor.
path(path&& p) noexcept;	Usado para mover (transferir a propriedade) de um caminho existente para um novo objeto std::filesystem::path. Isso é especialmente útil quando você deseja transferir um caminho de um local para outro de forma eficiente, sem a necessidade de cópia de dados.
path(string_type&& source, format fmt = auto_format);	Permite criar um objeto std::filesystem::path a partir de uma sequência de caracteres (por exemplo, uma string) movida. Isso é útil quando você tem uma string que representa um caminho e deseja convertê-la em um objeto de caminho. O parâmetro fmt permite especificar o formato do caminho, tornando-o flexível para diferentes formatos de string.
template <class Source> path(const Source& source, format fmt = auto_format);	Usado quando você tem uma fonte de dados que representa um caminho, como uma string, e deseja criar um objeto de caminho diretamente. A flexibilidade do parâmetro Source permite criar caminhos a partir de várias fontes de dados, como strings, iteradores e outros tipos de dados.
template <class InputIt> path(InputIt first, InputIt last, format fmt = auto_format);	Útil quando você tem um intervalo de caracteres representando um caminho e deseja criar um objeto de caminho a partir desses caracteres. Ele é especialmente útil quando você precisa manipular caminhos em partes, representadas por caracteres em um intervalo (por exemplo, ao ler caminhos de um arquivo de configuração).
template <class Source> path(const Source& source, const std::locale& loc, format fmt = auto_format);	Permite especificar uma std::locale para processamento de caracteres. Isso é útil quando você lida com caminhos que contêm caracteres especiais específicos do idioma e precisa garantir que eles sejam processados corretamente.
template <class InputIt> path(InputIt first, InputIt last, const std::locale& loc, format fmt = auto_format);	Semelhante à versão anterior, este construtor permite especificar uma std::locale para processamento de caracteres ao criar um objeto de caminho a partir de um intervalo de caracteres.

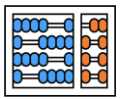


std::filesystem::file_status

- Usada para representar o estado de um arquivo ou diretório no sistema de arquivos.
- Permite inspecionar propriedades de arquivos ou diretórios sem lançar exceções.
- Útil para consultar informações antes de realizar operações específicas.

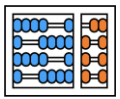
```cpp
#include <filesystem>
namespace fs = std::filesystem;

fs::path filePath = "example.txt";
fs::file_status status = fs::status(filePath);
```

```cpp
if (fs::is_regular_file(status)) {
    // É um arquivo regular.
} else if (fs::is_directory(status)) {
    // É um diretório.
} else {
    // Tipo não reconhecido.
}
```



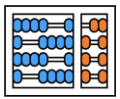
std::filesystem::file_type

Constante	Significado
none	indica que o status do arquivo ainda não foi avaliado ou ocorreu um erro ao avaliá-lo
not_found	indica que o arquivo não foi encontrado (isso não é considerado um erro)
regular	um arquivo regular
directory	um diretório
symlink	um link simbólico
block	um arquivo especial de bloco
character	um arquivo especial de caractere
fifo	um arquivo FIFO (também conhecido como pipe)
socket	um arquivo de soquete
implementation-defined	uma constante adicional definida pela implementação para cada tipo de arquivo adicional suportado pela implementação (por exemplo, a STL da MSVC define "junction" para junções NTFS)
unknown	o arquivo existe, mas seu tipo não pôde ser determinado



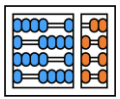
std::filesystem::perms

Membro Constante	Significado
none	nenhum bit de permissão está definido
owner_read	Proprietário do arquivo tem permissão de leitura
owner_write	Proprietário do arquivo tem permissão de escrita
owner_exec	Proprietário do arquivo tem permissão de execução/pesquisa
owner_all	Proprietário do arquivo tem permissão de leitura, escrita e execução/pesquisa (Equivalente a owner_read | owner_write | owner_exec)
group_read	O grupo de usuários do arquivo tem permissão de leitura
group_write	O grupo de usuários do arquivo tem permissão de escrita
group_exec	O grupo de usuários do arquivo tem permissão de execução/pesquisa
group_all	O grupo de usuários do arquivo tem permissão de leitura, escrita e execução/pesquisa (Equivalente a group_read | group_write | group_exec)
others_read	Outros usuários têm permissão de leitura
others_write	Outros usuários têm permissão de escrita
others_exec	Outros usuários têm permissão de execução/pesquisa
others_all	Outros usuários têm permissão de leitura, escrita e execução/pesquisa (Equivalente a others_read | others_write | others_exec)
all	Todos os usuários têm permissão de leitura, escrita e execução/pesquisa (Equivalente a owner_all | group_all | others_all
set_uid	Define o ID do usuário como o ID do proprietário do arquivo na execução
set_gid	Define o ID do grupo como o ID do grupo de usuários do arquivo na execução
sticky_bit	Significado definido pela implementação, mas o POSIX XSI especifica que, quando definido em um diretório, apenas os proprietários de arquivos podem excluí-los, mesmo que o diretório seja gravável para outros (usado com /tmp)
mask	Todos os bits de permissão válidos (Equivalente a all | set_uid | set_gid | sticky_bit)
unknown	Permissões desconhecidas (por exemplo, quando filesystem::file_status é criado sem permissões)



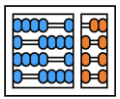
Diretórios

- Criar um Diretório:
  - `fs::create_directory("C:\\novo_diretorio");`
- Renomear um Diretório:
  - `fs::rename("C:\\antigo_nome", "C:\\novo_nome");`
- Mover um Diretório:
  - `fs::rename("C:\\origem\\diretorio", "C:\\destino\\diretorio");`
- Excluir um Diretório:
  - `fs::remove("C:\\diretorio_a_ser_excluido");`



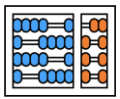
Diretórios

- Copiar um Diretório:
  - `fs::copy("C:\\origem\\diretorio", "C:\\destino\\diretorio_copia", fs::copy_options::recursive);`
- Alterar Permissão de Diretório:
  - `fs::permissions("C:\\diretorio", fs::perms::others_exec, fs::perm_options::remove);`
- Obter um Caminho Absoluto:
  - `fs::path caminhoAbsoluto = fs::absolute("C:\\relativo\\arquivo.txt");`
- Existência de um Diretório:
  - `bool arquivoExiste = fs::exists(caminho);`



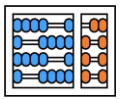
Diretórios

- Copiar um Diretório:
  - `fs::copy("C:\\origem\\diretorio", "C:\\destino\\diretorio_copia", fs::copy_options::recursive);`
- Alterar Permissão de Diretório:
  - `fs::permissions("C:\\diretorio", fs::perms::others_exec, fs::perm_options::remove);`
- Obter um Caminho Absoluto:
  - `fs::path caminhoAbsoluto = fs::absolute("C:\\relativo\\arquivo.txt");`
- Existência de um Diretório:
  - `bool arquivoExiste = fs::exists(caminho);`



std::filesystem::copy_options

Membro Constante	Significado
Opções que controlam copy_file() quando o arquivo já existe	
none	Reportar um erro (comportamento padrão)
skip_existing	Manter o arquivo existente, sem relatar um erro.
overwrite_existing	Substituir o arquivo existente.
update_existing	Substituir o arquivo existente somente se ele for mais antigo do que o arquivo sendo copiado.
Opções que controlam os efeitos de copy() em subdiretórios	
none	Pular subdiretórios (comportamento padrão).
recursive	Copiar recursivamente subdiretórios e seu conteúdo.
Opções que controlam os efeitos de copy() em links simbólicos	
none	Seguir links simbólicos (comportamento padrão).
copy_symlinks	Copiar links simbólicos como links simbólicos, não como os arquivos para os quais apontam.
skip_symlinks	Ignorar links simbólicos.
Opções que controlam o tipo de cópia feita por copy()	
none	Copiar o conteúdo do arquivo (comportamento padrão).
directories_only	Copiar a estrutura de diretórios, mas não copiar nenhum arquivo que não seja de diretório.
create_symlinks	Em vez de criar cópias de arquivos, criar links simbólicos apontando para os originais. Observação: o caminho de origem deve ser um caminho absoluto, a menos que o caminho de destino esteja no diretório atual.
create_hard_links	Em vez de criar cópias de arquivos, criar links rígidos que resolvam para os mesmos arquivos dos originais.

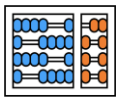


Iteradores em Diretórios

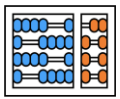
- directory_iterator
  - O directory_iterator é usado para iterar pelos itens de um diretório específico.
  - Ele fornece informações sobre cada item encontrado, como nome, tamanho e tipo.

- recursive_directory_iterator
  - O recursive_directory_iterator é usado para iterar recursivamente por todos os itens em um diretório e seus subdiretórios.
  - É útil para busca profunda de arquivos e diretórios

```
for (const fs::path& entry : fs::directory_iterator("C:\\diretorio")) {
    std::cout << "Nome do arquivo: " << entry.filename() << std::endl
              << "Tamanho do arquivo: " << fs::file_size(entry)
              << " bytes" << std::endl;
}
```



Serialização e Persistência



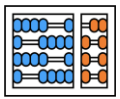
UNICAMP

Serialização

- Serialização é um mecanismo para converter um objeto em uma sequência de bytes.

- Essa sequência de bytes pode ser armazenada na memória ou transmitida por meio de uma comunicação.

- O processo reverso de serialização é chamado de desserialização, onde os dados na sequência de bytes são usados para reconstruir o objeto em sua forma original.



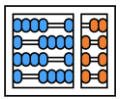
Persistência

- Em C++, o estilo tradicional de entrada e saída orientados a objetos (<< e >>) opera sobre tipos fundamentais em vez de objetos.

- No entanto, podemos usar a sobrecarga de operadores para trabalhar com entrada e saída de objetos de tipos definidos pelo usuário.

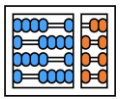
- Ao trabalhar com entrada e saída de objetos, estamos lidando com os dados do objeto, não com o objeto em si.

- A persistência de objetos, neste caso, se refere apenas aos dados sendo armazenados.



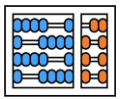
Persistência vs. Serialização

- Na persistência simples de dados do objeto, perdemos informações vitais sobre o objeto e seu tipo.
- Normalmente, armazenamos os atributos de um objeto, mas não detalhes como seu tipo.
- Como resultado, um programa que lê o objeto de volta do disco não terá informações sobre o tipo e não poderá reconstruir o objeto facilmente.
- Isso é especialmente problemático quando armazenamos objetos de tipos diferentes no mesmo arquivo.



Persistência vs. Serialização

- A serialização de objetos é diferente da persistência de dados do objeto.
- Um "objeto serializado" é representado como uma sequência de bytes.
- A informação na sequência de bytes inclui os dados dos atributos do objeto, seus tipos e o próprio tipo do objeto.
- Uma vez que essa informação de objeto serializado é gravada em um arquivo, ela pode ser lida, desserializada e usada para reconstruir o objeto na memória a qualquer momento.

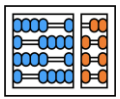


Manipulação de arquivos

- C++ fornece a biblioteca padrão <fstream> para manipulação **básica** de arquivos.

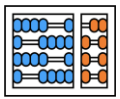
- Para operações já implementadas, consideramos bibliotecas de terceiros, como Boost.Filesystem e QT C++.

- Bibliotecas externas podem oferecer recursos poderosos e eficientes.



<fstream> (Biblioteca de Fluxo de Arquivos)

- Funcionalidades Principais:
  - Abertura, leitura e escrita de arquivos.
  - Trabalha com fluxos de entrada (ifstream) e fluxos de saída (ofstream).
  - Ideal para operações básicas de I/O de arquivos.
- Uso Típico:
  - Leitura e escrita de arquivos de texto simples.
  - Manipulação de arquivos de configuração.
- Diferenças Notáveis:
  - Trabalha principalmente com conteúdo de arquivo.
  - Não fornece operações diretas para manipular diretórios.

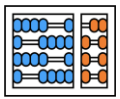


Abrindo um arquivo

```
void open(const char *filename);
```

```
void open(const char *filename, ios::openmode mode);
```

Modo	Descrição
ios::app	Modo de anexação. Todos os dados de saída são anexados ao final do arquivo.
ios::ate	Abre um arquivo para saída e move o controle de leitura/escrita para o final do arquivo.
ios::in	Abre um arquivo para leitura.
ios::out	Abre um arquivo para escrita.
ios::trunc	Se o arquivo já existir, seu conteúdo será truncado (apagado) antes de abrir o arquivo.
ios::binary	Usar o arquivo no modo binário.



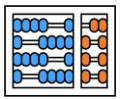
Fechando um arquivo

```
void close();
```

- Com o uso do fstream não é necessário fechar o arquivo

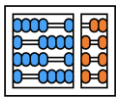
- O destrutor vai se encarregar de fechar o arquivo, apesar de ser uma boa prática fechá-lo explicitamente

Obs.: "Resource Acquisition Is Initialization" ou RAII é uma técnica de programação em C++ que associa o ciclo de vida de um recurso que deve ser adquirido antes de ser usado (memória alocada dinamicamente, thread de execução, soquete aberto, **arquivo aberto**, mutex bloqueado, espaço em disco, conexão de banco de dados - qualquer coisa que exista em quantidade limitada) à vida de um objeto. Isso ocorre a partir do Standard (27.8.1.2)



Lendo e escrevendo

- Similar ao uso aplicado com iostream (cin e cout)
- Esses operadores são “geralmente” usados para texto
- read e write são mais usados para o modo binário



UNICAMP

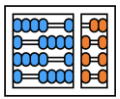
Lendo e escrevendo

```
istream& read (char* s, streamsize n)
```

```
ostream& write (const char* s, streamsize n);
```

- As operações read e write são usadas para realizar leitura e gravação de dados em um arquivo binário com a classe fstream em C++.

- Essas operações permitem que você leia e grave dados em formato binário, o que é útil para armazenar e recuperar estruturas de dados complexas, como objetos.

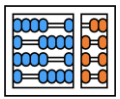


Buscando em arquivo (seekg e seekp)

```
istream& seekg (streampos pos);
ostream& seekp (streampos pos);
```

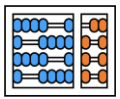
```
istream& seekg (streamoff off, ios_base::seekdir way);
ostream& seekp (streamoff off, ios_base::seekdir way);
```

Offset	Offset é relativo a...
ios_base::beg	Início do fluxo (começo do arquivo)
ios_base::cur	Posição atual no fluxo
ios_base::end	Fim do fluxo (final do arquivo)



Pontos de atenção

- Arquivo texto vs. Arquivo binário
- Persistências de objetos



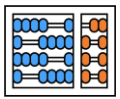
Arquivo texto vs Arquivo binário

Arquivo texto

- Arquivos de texto são legíveis por humanos.
- Eles armazenam dados como caracteres de texto.
- Dados numéricos são convertidos em texto antes de serem armazenados.
- Os dados podem ser lidos e interpretados diretamente por um humano.

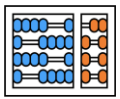
Arquivo Binário

- Arquivos binários são legíveis por máquinas.
- Eles armazenam dados em sua forma binária original.
- Não há conversão para texto; os dados são mantidos exatamente como estão.
- Os dados são mais compactos em arquivos binários, economizando, por exemplo espaço em disco.



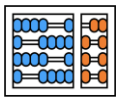
Escrevendo no arquivo

- Escrevendo o 13 em arquivo modo texto
  - 13 deve ser convertido para ASCII code. Ou seja:
    - 1 -> Código 49
    - 3 -> Código 51
  - Cada código é convertido para binário
    - 49-> 00110001
    - 51 -> 00110011
  - O que é escrito no arquivo
    - 0011000100110011



Escrevendo no arquivo

- Escrevendo o 13 em arquivo modo binário
  - Supondo que 13 é um inteiro (geralmente 4 bytes), ele é convertido para binário
    - 13-> 0000 0000 0000 0000 0000 0000 0000 1101
  - O que é escrito no arquivo
    - 00000000000000000000000000001101



Persistência de objetos

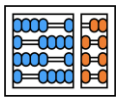
- Sobrecarga de operadores

Armazena dados:

```
ofstream & operator << (ofstream & ofs, Object o);
```

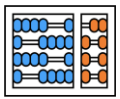
Recupera dados:

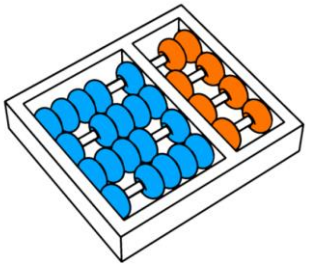
```
ifstream & operator >> (ifstream & ifs, Object & o);
```



Persistência de objetos

- Para se pensar:
  - Como serializar/desserializar objetos que fazem parte de herança?
  - E se os objetos contém ponteiros?
  - E se os ponteiros formarem um “ciclo”?





INSTITUTO DE COMPUTAÇÃO

UNICAMP

Prof. Dr. Bruno B. P. Cafeo

Sala 04
Instituto de Computação - Unicamp
Av. Albert Einstein, 1251
Cidade Universitária
Campinas – SP
13083-852

https://ic.unicamp.br/~cafeo/
cafeo@ic.unicamp.br