UNIVERSIDADE FEDERAL DE OURO PRETO
DECOM

Arthur Henrique Santos Celestino 21.1.4019
Caio Silas de Araujo Amaro 21.1.4111
Henrique Dantas Pighini 21.1.4025

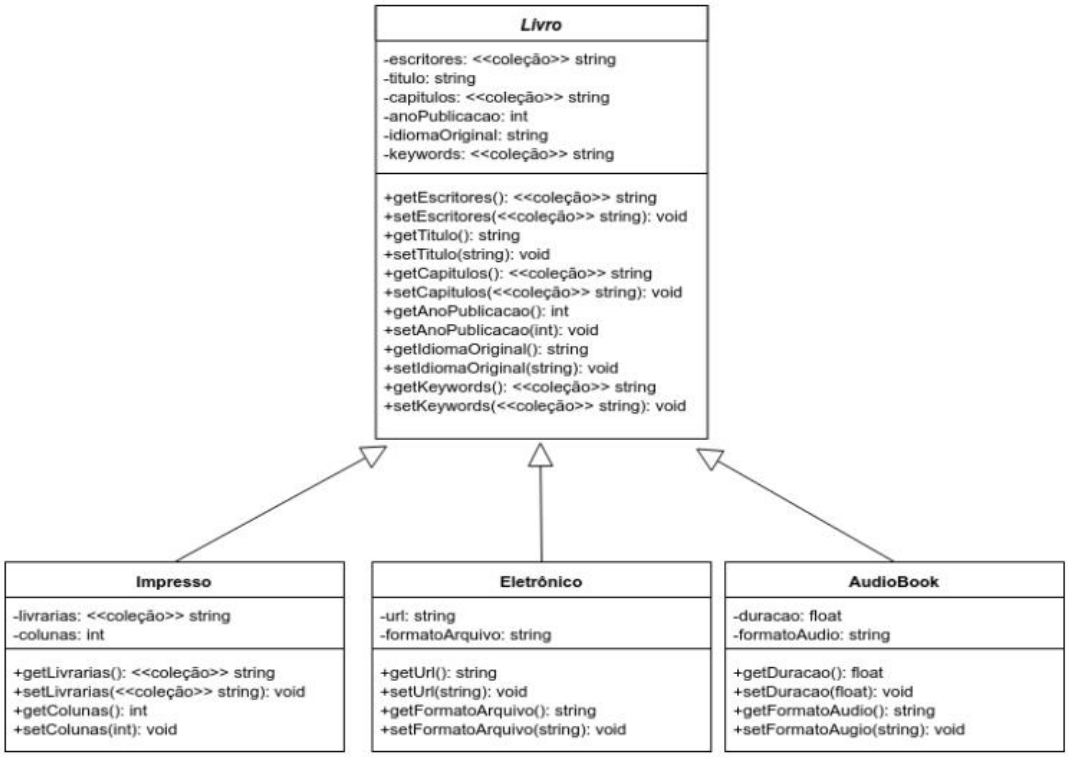
Trabalho Prático 1 - CPP

Ouro Preto - MG
2022

Introdução

A programação orientada a objetos tem como missão aproximar nossa lógica no momento de programar à forma que vemos e compreendemos as coisas do mundo. Para isso faz uso de classes e objetos, que englobam os dados, agrupando-os de maneira a facilitar nosso raciocínio.

O objetivo deste trabalho foi a implementação de uma biblioteca que tinha disponíveis livros em três versões: livro impresso, livro eletrônico e audiobook, que estavam distribuídos nas seguintes classes:



Durante a implementação do trabalho foi proposto pelo professor Guillermo Cámara Chávez que houvesse uma sobrecarga de operador em todas as saídas que pedissem para exibir os livros, entre outras especificações de saídas.

O sistema foi codificado no main. Onde contém todas as chamadas de operações descritas e implementadas e também contém uma coleção de objetos de livros impressos, eletrônicos e audiobooks, possibilitando a manipulação e transformação para futuras operações.

Para compilar o programa utilizamos a seguinte diretiva no terminal:

g++ main.cpp Eletronico.cpp Impresso.cpp AudioBook.cpp Livro.cpp Functions.cpp -o tp

./tp

Implementação

Containers usados

No trabalho, usamos diversos componentes da STL (*Standard Template Library)*. Ela é um conjunto de classes de template C + + para fornecer estruturas de dados de programação comuns e funções, como listas, pilhas, arrays, etc. É uma biblioteca generalizada e, portanto, seus componentes são parametrizados. Ela conta com alguns componentes chamados containers, que são estruturas conhecidas e já implementadas e testadas, desse modo se evita a repetição de código e também reduz a possibilidade de erros.

Utilizamos principalmente o container vector, pois além de ser o mais conhecido já previamente por nós (utilizamos a ideia de vetor desde introdução a programação), também é eficiente para se acessar elementos específicos contidos nele. Ele foi utilizado desde a criação das classes, até o main/funções.

```
class Livro{

private:
    vector <string> escritores;
    string titulo;
    vector <string> capitulos;
    int anoPublicacao;
    string idiomaOriginal;
    vector <string> keywords;
```

Primeiramente, criamos um vector de ponteiros para livros, onde em cada um desses ponteiros alocamos um objeto de alguma das classes derivadas, de acordo com o necessário. Dessa forma podemos facilmente mover toda nossa coleção de livros numa única estrutura.

```
vector <Livro*> biblioteca;
```

```
Eletronico *E = new Eletronico(escritores, titulo, capitulos, ano, idioma, keywords, url, formatoArquivo);
biblioteca.push_back(E);
```

Outra vantagem do vector que nos fez escolhê-lo, foi onde inserimos elementos, já que na grande maioria das vezes, apenas os colocamos no final, e no vector essa operação é O(1).

Em alguns momentos do código, também precisávamos acessar posições aleatórias específicas do vector, que é uma operação típica desse container, sem alto custo. Essa vantagem também nos beneficia no momento de ordenar elementos do vector, afinal com acesso direto a posições, as trocas de posições entre os elementos são mais rápidas.

```
vector <Livro*> idiomaCatch(vector <Livro*> biblioteca, string idioma){
    vector <Livro*> retorno;

    for(int i = 0; i < biblioteca.size(); i++){
        if(biblioteca[i]->getIdiomaOriginal() == idioma){
            retorno.insert(retorno.end(), biblioteca[i]);
        }
    }
```

Ao decorrer do código, fomos utilizando o container de maneira análoga à mostrada acima, sempre que precisamos utilizar nossa coleção de livros.

Outro container utilizado foi o multimap, para realizar o mapeamento dos livros pelo idioma. Ele foi escolhido ao invés do map pois permite repetição, e no nosso programa, existem mais de um livro com a mesma chave (nesse caso o idioma). Realizamos um typedef no início do código para evitar ficar reescrevendo a declaração a todo momento, já que é grande

```
typedef std::multimap<string, Livro*>Mmid;
```

```
Mmid mapeamento;

for(int i = 0; i < biblioteca.size(); i++){
    if(biblioteca[i]->getIdiomaOriginal() == "Ingles"){
        mapeamento.insert(Mmid::value_type(English, biblioteca[i]));
    }
    if(biblioteca[i]->getIdiomaOriginal() == "Espanhol"){
        mapeamento.insert(Mmid::value_type(Espano, biblioteca[i]));
    }
    if(biblioteca[i]->getIdiomaOriginal() == "Frances"){
        mapeamento.insert(Mmid::value_type(Francais, biblioteca[i]));
    }
    if(biblioteca[i]->getIdiomaOriginal() == "Portugues"){
        mapeamento.insert(Mmid::value_type(Portuguese, biblioteca[i]));
    }
}
```

Algoritmos usados

Outro componente importante da STL são os algoritmos. Eles podem ser utilizados genericamente, em vários tipos de contêineres. Os algoritmos operam indiretamente sobre os elementos de um container usando iteradores. Eles fornecem os meios pelos quais se executa a inicialização, classificação, pesquisa e transformação do conteúdo dos containers.

Utilizamos neste trabalho o algoritmo sort. Escolhemos o sort pois em algumas funções precisávamos ordenar os dados.

```
sort(retorno.begin(), retorno.end(), [](Livro* lv1, Livro* lv2){return lv1->getEscritores()[0] < lv2->getEscritores()[0];});
```

Outro motivo da escolha está no fato de evitar repetição de código, já que sem ele precisamos implementar algum algoritmo de ordenação como mergesort, quicksort, etc, ainda podendo gerar erros. Já que o algoritmo é testado e seguro, serviu para nossos propósitos.

Análise de containers

● Acessar uma posição específica de um contêiner:

Para o acesso a posição específica o ‘vector’ é o melhor pois permite acesso direto via índice e também suportam iteradores de acesso aleatório.

● Adicionar um elemento e manter somente elementos únicos no contêiner:

Para adicionar e um elemento e manter o ‘set’ é o melhor, pois armazena um conjunto de chaves sem repetição e sua complexidade logarítmica e de O(log n).

● Inserção no final;

Para a inserção no final o melhor é ‘vector’ pois sua inserção é sempre no final.

● Remoção no final:

Para a remoção no final o melhor método é o ‘vector’ pois a remoção e a inserção dele possuem tempos lineares.

● Retornar um valor baseado em uma chave específica (não necessariamente inteiros):

Para se retornar um valor em uma chave específica a melhor é a ‘map’ pois contém pares de valores-chave com chaves exclusivas, e sua complexidade da pesquisa para std::map é $O(log\ n)$ (logarítmica no tamanho do container).

● Inserção no início:

Para a inserção no início teremos como melhor opção a ‘deque’ (fila de duas extremidades), pois permite rápidas inserções e exclusões no início e no final do container. Eles são semelhantes aos vetores, mas são mais eficientes no caso de inserção e exclusão de elementos.

● Remoção no início:

Para a remoção no início teremos como melhor escolha ‘deque’ (fila de duas extremidades), pois são filas em que as operações de inserção e exclusão são possíveis em ambas as extremidades.

● Busca por um elemento:

Para buscar um elemento teremos duas que são ‘set’ e ‘map’, pois realizam a pesquisa em tempo $O(log\ n)$, em comparação com vector e list, que realizam a mesma pesquisa em tempo $O(n)$, ou seja, set e map são melhores para realizar a busca de um elemento.

● Contêiner com o comportamento de primeiro a entrar é o último a sair:

Para esse tipo de comportamento temos a ‘stack’ que funciona como uma pilha, ou seja, o último elemento a ser inserido, será o primeiro a ser retirado.

● Contêiner com o comportamento de primeiro a entrar é o primeiro a sair.

Para o comportamento onde o primeiro a entrar é o primeiro a sair o ‘queue’ é o melhor pois são estruturas de dados do tipo FIFO (first-in, first-out), onde o primeiro elemento a ser inserido, será o primeiro a ser retirado, ou seja, adiciona-se itens no fim e remove-se do início.

Conclusão

Dessa forma, podemos concluir diversas coisas sobre o paradigma da orientação a objetos com esse trabalho. Nele vimos sobre diversos aspectos da STL, e sua utilidade para evitar erros, repetição de código e facilitação de manipulação de dados. Com os containers conseguimos armazenar e trabalhar sobre nossas informações de maneira muito fácil, utilizando os iteradores para nos movermos sobre ele.

Por fim, o presente trabalho foi muito proveitoso a todo grupo, pois nos permitiu exercitar um pouco alguns dos conceitos vistos na aula(classes, herança, sobrecargas, containers, iterators, algoritmos, etc , já que apenas com a prática se aprendem certos detalhes da linguagem.