Implementação de uma caixa eletrônica em C++

Daniel Becker, Ernesto Rodríguez

13 de novembro de 2020

1 Introdução

Este relatório se refere ao trabalho final da matéria Linguagens de Programação, instruída durante o Período Letivo Especial (PLE) para os alunos da Universidade Federal do Rio de Janeiro (UFRJ). O relatório tem como objetivo apresentar o programa desenvolvido, sua implementação e os seus casos de uso. Os códigos desenvolvidos foram disponibilizados, de forma aberta, em Github[1].

1.1 Objetivo do programa

O objetivo do programa é emular um sistema básico de caixa eletrônico, tomando como referência aqueles presentes em supermercados, usando duas linguagens de programação, $C++$ e SQL. O programa simula como acontecem as transações em supermercados e lojas. Dessa forma, são providenciadas funções que fazem a interface entre o estoque da nossa loja fictícia (modelado como um banco de dados) e nosso terminal de caixa. O projeto é esquematizado na Figura 1, onde podemos ver que o ambiente em $C++$ realiza diversas tarefas, sendo um intermediador entre a linguagem SQL e o usuário.

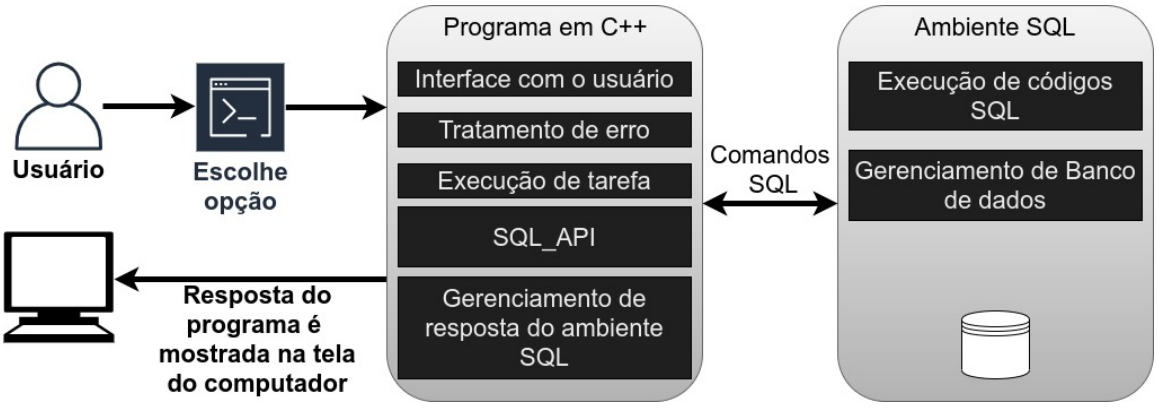

Figura 1: Esquema do projeto proposto.

[1] https://github.com/ernestorodg/trabalhoFinalLingProg

1.2 Sobre o programa

O programa implementa um caixa eletrônico que realiza comunicações com um banco de dados local. Ao comprar um produto, um programa deve ser capaz de retirar esse produto do estoque, automatizando assim muitas tarefas de gerenciamento. Além disso, uma caixa deve ser capaz de cancelar uma compra que tenha sido feita. Por outro lado, cada vez que um produto é aceito na caixa, é necessário atualizar o saldo. A mesma coisa acontece caso o valor referente a um produto deva ser retirado do caixa. Portanto, nosso programa implementa todas estas funções e as devidas verificações que devem ser feitas caso o usuário introduza valores não correspondentes aos pedidos.

Simulando um mercado, uma ou mais caixas devem ser capazes de interagir com um banco de dados. Dessa forma, a escolha da linguagem SQL é benéfica para esta aplicação, possibilitando o uso de funções já pré-configuradas que facilitem o gerenciamento de um banco de dados.

A proposta principal do trabalho é mostrar como os alunos podem executar comandos de outras linguagens usando C++. Dessa forma, o nosso programa propõe algumas funções que realizam uma interação com arquivos *.sql* já configurados, simulando funções em outras linguagens. Uma tabela SQL é criada com antecedência, já que isto varia de projeto para projeto.

1.3 Observações

É importante mencionar que o trabalho teve um desvio do proposto inicialmente. Na proposta do trabalho é dito que seria usada a ferramenta SQLAPI++ para a implementação. No entanto, esta ferramenta se mostrou pouco amigável para a nossa proposta. Além disso, muitas das funcionalidades da ferramenta não seriam usadas no nosso projeto. Portanto, foi decidido usar a biblioteca "sqlite3.h"para realizar a interface com a linguagem *SQL*.

2 Implementação do programa

2.1 Classes implementadas

Abaixo são descritas as classes que foram implementadas para o nosso trabalho.

2.1.1 DataBank

A classe DataBank implementa um gerenciador de banco de dados SQL. Ela possui um construtor que irá receber o nome do banco de dados SQL que irá ser aberto. Caso o nome não exista, o banco de dados é criado. Ela tem como atributos:

1. sqlite3* dataBank: Ponteiro para um objeto da estrutura sqlite3. Esta estrutura é definida na biblioteca sqlite3.h. A estrutura possibilita o chamado de funções que interagem com a linguagem SQL e o banco de dados criado;
2. Records records: É um vetor de vetores de *strings*. Este atributo guarda a informação do banco de dados, caso seja necessário.

Além disso, DataBank implementa os seguintes métodos:

1. static int extractDataBank(void*, int, char**, char**): Extrai do banco de dados toda a informação, sendo atribuída ao vetor records;
2. void setDataBank(const char *): Abre o arquivo banco de dados. Caso o valor recebido no ponteiro para caracter não corresponda a nenhum banco de dados no diretório, é criado um novo banco de dados;

3. int createTable(): cria uma tabela para o banco de dados. Esta função é preconfigurada, sendo escrita no arquivo "create_table.sql", já que depende de cada projeto. Caso ocorra algum erro, retorna -1;

4. int insertOnDataBank(std::string , std::string , std::string , std::string , std::string): insere informação no banco de dados. Caso ocorra algum erro, retorna -1;

5. int showDataBank(): mostra o conteúdo do banco de dados;

6. int deleteRowFromDataBank(int): recebe um ID identificando uma determinada linha da tabela usada. Apaga essa linha. Caso ocorra algum erro, retorna -1;

7. int closeDataBank(): fecha o arquivo banco de dados. Caso ocorra algum erro, retorna -1;

8. int updateDataBank(int, int, std::string): atualiza o banco de dados. No primeiro argumento recebe a coluna que será atualizada entre as disponíveis. No segundo argumento recebe a ID da linha que irá ser atualizada. No terceiro argumento recebe o novo valor. Retorna -1 caso ocorra algum erro. Retorna 0 caso não ocorra nenhum erro;

9. int getFirstEmptyID(): retorna a primeira ID do banco de dados que possa ser usada. Isto facilita operações como inserção;

10. int updateRecords(): atualiza o atributo *records* com os valores do banco de dados, para caso haja alguma atualização que deva ser modificada;

11. int lookInRecords(int): procura um determinado valor no atributo *records*;

12. void showPropertiesByID(int): mostra as colunas de uma determinada linha, segundo o ID da linha;

13. int getProductPrice(int): retorna o preço de um produto do nosso programa;

14. int getProductAmount(int): retorna a quantidade disponível de um produto do nosso programa;

15. int lookForProduct(std::string): procura um determinado produto no banco de dados. Caso o produto não exista, retorna -1. Caso o produto exista, retorna 0;

16. std::string openSqlCmd(std::string): abre um arquivo e retorna o que está escrito nele;

17. Records getRecords(): retorna o atributo *records*;

Cabe ressaltar que algumas funções, referentes a "produtos", são particulares do nosso programa, já que varia segundo a Tabela SQL que tenha sido criada.

2.1.2 Caixa

A classe Caixa implementa uma caixa de compras. A classe possui um construtor, que recebe o saldo inicial da caixa. Caso não seja passado nenhum saldo, o saldo é inicializado com um valor fixo arbitrário. Além disso, a classe Caixa possui um destrutor que fecha o arquivo de banco de dados que esteja sendo usado.

Atributos:

1. double saldo: guarda o saldo de uma compra;

2. DataBank bancoDados: objeto da classe DataBank. Ver Subseção 2.1.1;

3. std::vector<std::vector<int» items: objeto da classe *vector*. Armazena os itens que foram comprados e a quantidade. Isto permite operações de cancelamento.

Métodos:

1. double getSaldo(): retorna o valor do atributo *saldo*;
2. void setSaldo(double): atribui um valor ao atributo *saldo*;
3. int getProduct(std::string): realiza a função de vender um produto. Isto é, tem como função adicionar a *saldo* o valor do produto caso a compra tenha sido sucedida. Além disso, consulta o banco de dados e realiza testes, caso, por exemplo, não haja suficiente estoque. Retorna 0 caso não haja ocorrido nenhum erro. Retorna -1 caso haja ocorrido algum erro. Caso a compra tenha sido bem sucedida, os dados do produto são salvos no atributo *items* caso seja necessário realizar alguma tarefa com eles, como por exemplo, cancelar o item;
4. int cancelLastProduct(): cancela o último produto que foi passado à função *getProduct()*. A informação salva é retirada do atributo *items* e com ela são atualizados os atributos *saldo* e banco de dados;
5. void executeDataBaseCommand(int): função auxiliar para executar comandos relacionados à classe DataBank;

2.1.3 Interface

A classe Interface facilita ter uma interface com o usuário mais amigável. Ela usa códigos de escape ANSI[2] para realizar tal tarefa. A classe não possui atributos, somente métodos estáticos, já que não há necessidade de criar um objeto Interface para realizar tais tarefas.

1. static void cleanScreen(): apaga todos os caracteres do terminal;
2. static void saveScreen(): salva todos os caracteres presentes no terminal;
3. static void restoreScreen(): imprime todos os caracteres salvos no método *saveScreen()*;
4. static void setCursorToBegin(): coloca o cursor do terminal para a posição <0;0>. Isto é, a esquina superior esquerda do terminal;
5. static void saveCursorPosition(): salva a posição do cursor no terminal;
6. static void restoreCursorPosition(): coloca o cursor do terminal à posição salva pelo método *saveCursorPosition()*;

2.1.4 Menu

A classe Menu oferece alguns métodos para facilitar a interação com o usuário e a verificação de entradas. A classe possui um único atributo para armazenar a entrada que foi salva anteriormente.

1. void showCashierMenu(): Mostra as opções disponíveis para interagir com os métodos da classe Caixa 2.1.2;
2. void showDataBankMenu(): Mostra as opções disponíveis para interagir com o banco de dados;
3. int defineNumericInput(): verifica uma entrada numérica. Retorna -1 caso a entrada não tenha sido numérica. Caso não ocorra nenhum erro, retorna a própria entrada;

[2] https://en.wikipedia.org/wiki/ANSI_escape_code

4. std::string defineLiteralInput(): verifica uma entrada que deveria ser uma *string*. Retorna a própria *string*, caso a verificação tenha sido realizada com sucesso. Retorna uma *string* vazia (), caso tenha ocorrido algum erro;

3 Casos de Uso

A classe DataBank foi projetada para interagir com um banco de dados. Dessa forma, ela pode ser usada fora do nosso programa. Já o resto das classes tem um uso menos genérico, sendo mais restritos à aplicação que foi dada no nosso projeto.

3.1 Menu da caixa

Uma vez executado o programa, é mostrado um menu estilo caixa de supermercado, como visto na Figura 2. Nesta tela, podemos digitar uma quantidade de um produto e o nome do produto para realizar a compra. Caso seja digitado somente o nome do produto, a quantidade será interpretada como 1. Assim, caso a quantidade do produto exista no banco de dados, este e o saldo da loja são atualizados. Para sair do programa, basta digitar a opção "EXIT".

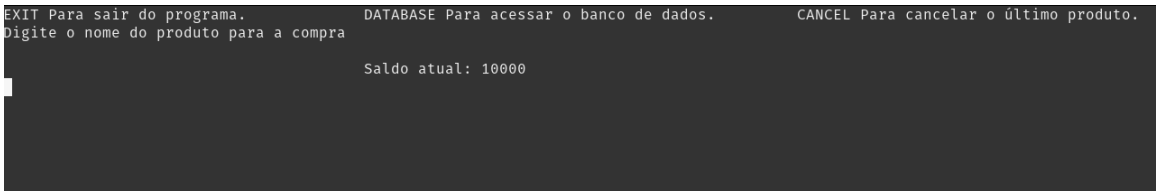

Figura 2: Interface da caixa da loja.

Outra opção que o usuário tem neste menu é cancelar o produto anterior. Caso o produto seja cancelado, o saldo e o banco de dados são atualizados com os valores anteriores. Se o usuário tentar cancelar um produto sem ter havido nenhuma compra, um erro será retornado e uma mensagem será mostrada ao usuário.

Ainda nesta interface, o usuário poderá acessar o banco de dados digitando "DATABASE". Este comando levará o usuário ao menú do banco de dados, explicado na Subseção 3.2.

3.2 Menu do banco de dados

Caso o usuário digite "DATABASE"no menú da caixa (Subseção 3.1), serão mostradas na tela as opções que podem ser executadas relacionadas ao banco de dados. A interface é mostrada na Figura 3.

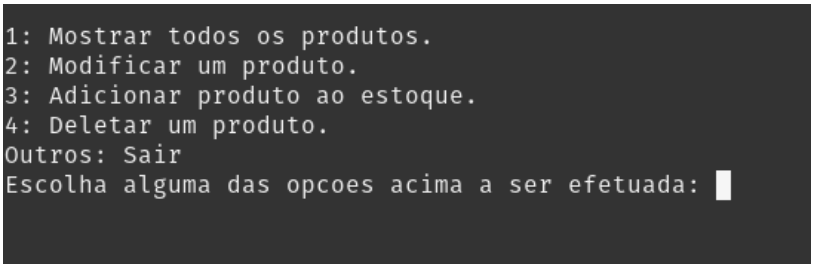

Figura 3: Interface do banco de dados.

Se nesta interface, digitamos a Opção 1, por exemplo, será mostrado o conteúdo do banco de dados, como visto na Figura 4. O resto das opções não mostram nada além de uma mensagem de confirmação ou erro.

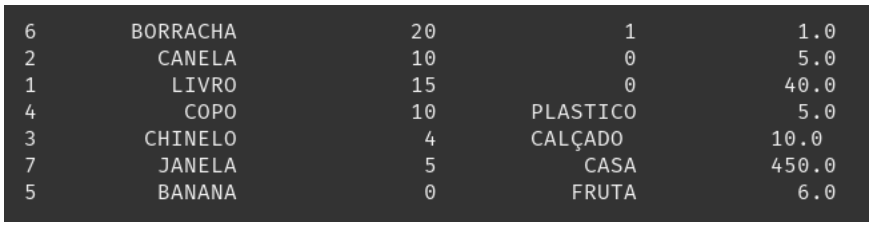

Figura 4: Mensagem após escolher a opção 1 na interface do banco de dados.

4 Conclusão

Este relatório apresentou o programa implementado pelo nosso grupo para simular um caixa de mercado, com o uso de funções da linguagem SQL. A biblioteca usada, sqlite3, se mostrou de grande ajuda para realizar as interações entre as linguagens $C++$ e SQL. O trabalho permitiu ao grupo mostrar os conhecimentos adquiridos durante o PLE da UFRJ, além de incentivar a procura por conhecimento relativo à interação entre linguagens diferentes.

Tendo em consideração que a aplicação no mundo real é vista como várias caixas acessando um mesmo banco de dados, a nossa escolha de usar um banco de dados SQLite se mostrou muito adequada. A junção de $C++$ e SQLite foi satisfatória para atender o nosso projeto. Assim, acreditamos que aplicações reais baseadas no nosso modelo podem ser de alta utilidade.

A versão final do programa corresponde ao projeto previamente proposto. No entanto, o programa ainda pode ser melhorado. Por exemplo, algumas melhorias são: a atribuição de uma interface de dados mais robusta; implementar a possibilidade de uso através de uma rede de computadores; etc. Assim, podemos concluir que o programa tem potencial para se tornar uma aplicação mais robusta, a partir da nossa implementação.