Programação Orientada à Objetos:

Trabalho Prático 1:

Introdução:

O intuito do trabalho é programar a interface e as funções de um banco, separando em módulos e fazendo uso de herança. O código foi separado em módulos: Cliente, Movimentação, Conta, Banco e Interface. As classes são divididas entre de negócio (Conta, Movimentação e Cliente), em que serão feitas as definições base para o banco. A classe de gerência (Banco), em que é feita as funções para as operações bancárias e todas as funções relacionadas a interface. A classe interface é destinada para a interação com o cliente (todas as entradas do teclado e para printar na tela).

Como Compilar e executar:

O código foi programado no sistema operacional Microsoft Windows e a IDE utilizada foi a Microsoft Visual Studio, a qual também foi usada para compilar o programa. Caso prefira baixar pelo prompt de comando. Insira a linha:

g++ –o banco main.cpp banco.cpp movimentacao.cpp conta.cpp interface.cpp

A qual irá gerar um executável com o nome “banco”, após isso insira a linha:

./banco

Após isso o terminal irá compilar o programa.

Implementação do Código:

Header file:

· Cliente.h

A cliente.h foi o primeiro header file criado e nele contém a classe cliente onde está inserido as variáveis privadas: nomeCliente, cpf_cnpj, endereço, fone. Como mostrado na figura abaixo:

```
private:
    string nomeCliente;
    string cpf_cnpj;
    string endereco;
    string fone;
```

Além disso, na parte pública da classe está presente o construtor default e o construtor que seta todos os atributos do cliente, e também um operador para atribuir um Cliente. E por fim tem-se os métodos setters e getters para todas as variáveis do cliente e o seu destrutor como demonstrado abaixo:

```
public:
    Cliente();
    Cliente(string n, string c, string e, string f);
    void operator =(const Cliente& c);
    string getNome();
    string getCpf_cnpj();
    string getEndereco();
    string getFone();
```

```
    void setNome(string n);
    void setCpf_cnpj(string c);
    void setEndereco(string e);
    void setFone(string f);
    ~Cliente();
};
```

·Movimentação.h

A movimentação.h foi a segunda classe criada nele tem-se a classe movimentação, onde está inserida as variáveis privadas: dataMov(um vetor de string para armazenar a data, sendo a posição 0 o dia, a posição 1 o mês e a posição 2 o ano) e as friends classes da Conta:

```
private:
vector<string> dataMov;
string descricao;
char debitoCredito;
double valor;
friend class Conta;
friend class Cliente;
friend class Banco;
```

Na parte pública da classe tem os construtores e os getters e setters das variáveis da privadas e o seu destrutor.

```
public:

Movimentacao(string d, char op, double v);

Movimentacao(string d, char op, double v, string ano, string mes, string dia);

vector<string> getDataMov();

string getDescricao();

char getOp();

double getValor();

void setDescricao(string descr);

void setOp(char db);

void setValor(double v);

~Movimentacao();
```

Conta.h

A classe conta está no header contar.h e dentro dessa classe estão as variáveis privadas numConta, saldo, cliente (variável do tipo cliente), uma lista do tipo Movimentacao com o nome de movimentações e as classes amigas dessa.

```
private:

int numConta;

double saldo;

Cliente cliente;

list<Movimentacao> movimentacoes;

friend class Movimentacao;

friend class Banco;

friend class Cliente;
```

Na parte pública da classe tem os construtores e os getters e setters das variáveis da privadas e o seu destrutor, além da função que gera o extrato.

```
public:

static int proximoNumConta;

Conta(Cliente *c);

Conta(int nconta, double sald, Cliente *c, list<Movimentacao> mov);

int getNumConta();

double getSaldo();

Cliente* getCliente();

bool debitar(double v,string d);

bool debitar(double v,Movimentacao mov);

void creditar(double v, string d);

void print();

list<Movimentacao> extrato();

list<Movimentacao> extrato(vector<string> di);

list<Movimentacao> extrato(vector<string> di, vector<string> df);
```

Banco.h

Nesse último header file na parte privada tivemos o atributo do nome do banco(nomeBanco), a lista de cliente (listaClientes) e as classes amigas, como demonstrado abaixo:

```
private:
  string nomeBanco;
  list<Cliente> listaClientes;
```

```
list<Conta> listaContas;
friend class Cliente;
friend class Conta;
friend class Movimentacao;
```

Na parte pública Temos o construtor do banco os setters para cliente e conta e as funções solicitadas no roteiro e as funções responsáveis por ler e gerar arquivos.

```
public:
Banco(string nomeBanco);
void setCliente(Cliente c);
void setConta(Conta c);
void operator =(const Banco& c);
void delCliente(int num);
void delConta(int num);
void deposito(int nconta, double valor);
void saque(int nconta, double valor);
void saque(int nconta, double valor, string ano, string mes, string dia);
void transferencia_conta(int conta_origem, int conta_destino,double valor);
void tarifa();
void debitar_cpmf();
void saldo(int nconta);
void criar_conta(Cliente c);
void excluir_conta(int nconta);
list <Cliente> get_clientes();
list <Conta> get_contas();
void gravar_dados();
void ler_dados();
```

Construtores:

Na classe cliente tem dois construtores, o primeiro é construtor default e o segundo atribui por parâmetro o nome do cliente, cpf/cnpj, endereco e telefone. Como demonstrado na figura abaixo:

```
Cliente::Cliente(){};
Cliente::Cliente(string n, string c, string e, string f){
   nomeCliente = n;
   cpf_cnpj = c;
   endereco = e;
   fone = f;,
}
```

Além disso, também tem o destrutor do cliente:

```
Cliente::~Cliente(){}
```

Na classe Movimentacao tem dois construtores, o primeiro atribui por parâmetro a descrição, o tipo de operação e o valor. Além de gerar a data da movimentação em tempo real, utilizando a biblioteca <ctime> do c++. O segundo construtor permite criar uma movimentação atribuindo uma data específica para ela, não sendo necessariamente em tempo real.

```
   Movimentacao(string d, char op, double v);
   Movimentacao(string d, char op, double v, string ano, string mes, string dia);
```

E o seu destrutor

```
   ~Movimentacao();
```

Na classe conta tem dois construtores, o primeiro que inicializa a conta com um cliente, e o segundo que atribui por parâmetro o número da conta o saldo o cliente e a lista de movimentação.

```
Conta(Cliente c);
Conta(int nconta, double sald, Cliente c, list<Movimentacao> mov);
```

Na classe banco tem um construtor que é responsável pela atribuição do nome do banco.

Sobrecarga de Operadores:

No nosso código utilizamos o operador ‘=’ para atribuição. O qual foi utilizado 2 vezes no programa. Na classe cliente e na classe banco.

Operador na classe cliente:

```
void Cliente::operator =(const Cliente& c){
    this->nomeCliente = c.nomeCliente;
    this->cpf_cnpj = c.cpf_cnpj;
    this->endereco = c.endereco;
    this->fone = c.fone;
}
```

onde atribui todos os atributos do cliente que é passado por referência a variável do tipo cliente que chama a função.

Operado na classe banco:

```
void Banco::operator =(const Banco& c){
  this->nomeBanco = c.nomeBanco;
  this->listaClientes = c.listaClientes;
  this->listaContas = c.listaContas;
}
```

em que atribui o nome do banco à sua lista de clientes e a lista de contas, da variável que é passada por referência para a variável que chama a função.

Getters e setters básicos:

Como todas as nossas variáveis presentes nas 4 classes são privadas utilizamos métodos getters e setters para todas as variáveis do programa, pois assim seria possível mudar o atributo delas, e também utilizar o seu valor para alguma operação, ou comparação.

Na classe cliente.h foi usado o método set em todas as variáveis para que se possa alterar de forma mais rápida alguma variável privada. Esse método é válido pois em um cenário de um banco, os clientes podem mudar o endereço e o telefone ao longo do tempo, utilizando as funções de set, podemos alterar essas informações com uma linha de código.

Na classe movimentacao.h não foi necessário utilizar esse método pois o objetivo da mesma é agrupar as operações bancárias com o sua respectiva data. Essa função não é necessária modificar as variáveis pois o histórico de movimentações é único e imutável.

Na classe conta.h é utilizado as funções get para retornar o número da conta, o saldo e o cliente. Essas funções são importantes para a implementação da classe

banco.h pois o número da conta e o cliente são usadas constantemente. A função get para o saldo é importante na interface, para que o usuário tenha idéia do valor contido na conta.

Na classe banco.h as funções set para cliente e conta são necessárias para adicioná-los na no container list. Já as funções get cliente e conta são necessárias para retornar um container list com todos os clientes/contas. Essa parte é importante em outras funções como excluir_conta() e excluir_cliente(), e para a função de gravar_dados().

Main:

Como todas as operações são feitas na classe interface, o arquivo main.cpp tem como o objetivo basicamente de declarar o nome do Banco e para criar o construtor da interface.

Ler e escrever arquivos:

São as duas funções mais importantes do programa, pois sem elas não tem como realizar o sistema com o mínimo de realidade. A função ler_dados() é chamada uma vez somente logo no início do código, para que todos os valores contidos no arquivo Bancodedados.txt sejam setados na Classe Banco.

```
void Banco::ler_dados(){
  ifstream in("Bancodedados.txt");
  string linha;
  getline(in,linha);
  while (!(in.eof())) {
    if(linha == "Clientes"){
      getline(in,linha);
      while(linha != "Conta"){
        string nome = linha;
        getline(in,linha);
        string cpf = linha;
        getline(in,linha);
        string endereco = linha;
        getline(in,linha);
        string fone = linha;
        Cliente a (nome,cpf,endereco,fone);
        this->setCliente(a);
        getline(in,linha);
      }
    }
    else if(linha == "Conta"){
      getline(in,linha);
      int nconta = stoi(linha);
      getline(in,linha);
      double sald = stod(linha);
      getline(in,linha);
      string ncli = linha;
      list<Cliente> c = get_clientes();
      Cliente aux;
```

```
    for (auto cli = c.begin() ; cli != c.end() ; cli++){
      if(ncli == cli->nomeCliente){
        aux.setNome(cli->nomeCliente);
        aux.setCpf_cnpj(cli->cpf_cnpj);
        aux.setEndereco(cli->endereco);
        aux.setFone(cli->fone);
        break;
      }
    }
    list<Movimentacao> m;
    while(linha != "Conta"){
      getline(in,linha);
      while((linha != "mov")){
        string ano = linha;
        getline(in,linha);
        string mes = linha;
        getline(in,linha);
        string dia = linha;
        getline(in,linha);
        string d = linha;
        getline(in,linha);
        char op = linha[0];
        getline(in,linha);
        double v = stod(linha);
        Movimentacao a(d,op,v,ano,mes,dia);
        m.push_back(a);
        getline(in,linha);
        if(linha == "Conta"){break;}
        if(linha.size() == 0){break;}
      }
      if(linha.size() == 0){break;}
    }
    Conta co(nconta,sald,aux,m);
    this->setConta(co);
  }
 }
 in.close();
}
```

O ponto chave dessa função é a utilização de um parâmetro auxiliar para setar o Banco, a Conta, o Cliente e a movimentação. Isso é necessário pois no momento de gravar cada parâmetro é adicionado em cada linha, facilitando assim a leitura (a função getline ler toda a string de uma linha).

A gravação de dados é feita de forma sequencial, primeiramente é inserido os Clientes, enviando os parâmetros nome,cpf_cnpj,endereço e telefone, exatamente nessa ordem. Em seguida é enviado o valor de cada conta, com os parâmetros número da conta, valor, nome do cliente, histórico de movimentações(ano,mês,dia,descrição, tipo, valor), exatamente nessa ordem.

```
void Banco::gravar_dados(){
 ofstream out("Bancodedados.txt");
 out<<"Clientes"<<endl;
 list<Cliente> c = get_clientes();
 for (auto j = c.begin() ; j!= c.end(); j++){
   out<<j->nomeCliente<<endl;
   out<<j->cpf_cnpj<<endl;
   out<<j->endereco<<endl;
   out<<j->fone<<endl;
```

```
}
list<Conta> co = get_contas();
for (auto j = co.begin() ; j != co.end() ; j++){
  out<<"Conta"<<endl;
  out<<j->numConta<<endl;
  out<<j->saldo<<endl;
  Cliente clienteaux = j->cliente;;
  string stringaux = clienteaux.getNome();
  out<<stringaux<<endl;
  for(auto k = j->movimentacoes.begin(); k != j->movimentacoes.end(); k++){
    out<<"mov"<<endl;
    out<<k->dataMov[0]<<endl;
    out<<k->dataMov[1]<<endl;
    out<<k->dataMov[2]<<endl;
    out<<k->descricao<<endl;
    out<<k->debitoCredito<<endl;
    out<<k->valor<<endl;
  }
}
out.close();
}
```

Conclusão:

Durante o desenvolvimento do projeto teve etapas em tivemos muita dificuldade. Como na parte de ler e escrever arquivos e na classe da interface. Para ler e escrever arquivos tivemos que pivotar diversas vezes a idéia para que ficasse o mais simples possível. A classe interface é a classe mais trabalhosa do projeto, pois requer um esforço para conseguir linkar todas as classes e ao mesmo tempo apresentar um layout minimamente aceitável para o cliente.

Uma ferramenta que nos ajudou bastante para que o projeto fluir bem foi o GitHub em conjunto com o GitKraken. Por meio dessa plataforma podemos controlar de maneira muito eficaz o versionamento do código e facilita o trabalho em conjunto.