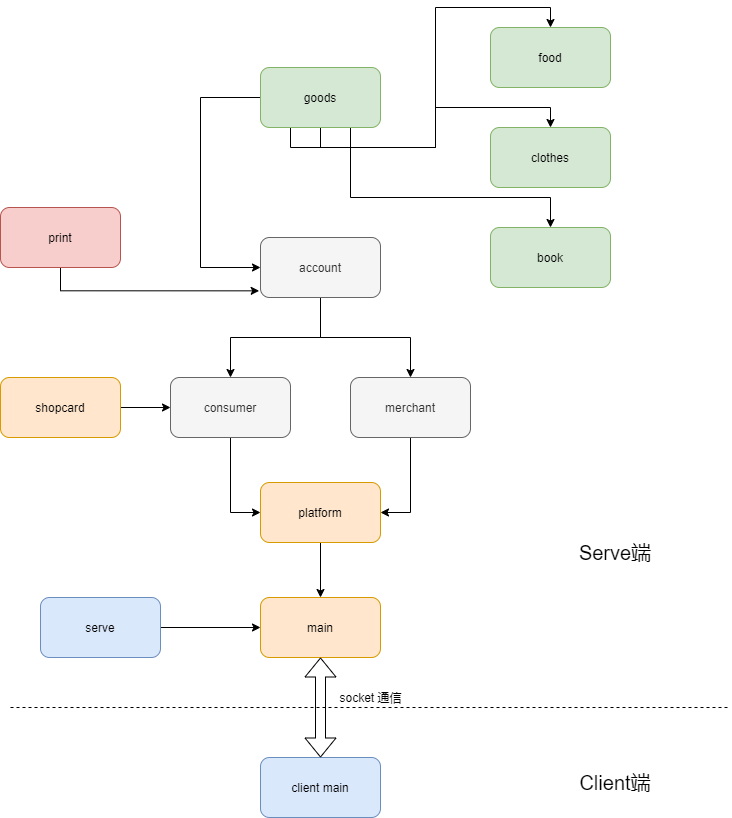
电商交易平台实验报告

姓名：胡敏臻 班级：2019211309 学号：2019211424

1. 任务描述

模拟电商交易平台。在该平台中，在不登录的情况下可以查看、筛选商品。在登录的情况下可以修改密码，管理余额等。作为商家可以添加、管理商品，给商品打折；在实验二中，作为消费者可以选择商品添加管理购物车、生成取消或支付订单进行购物。此外，在实验三中还可以实现在客户端和服务端进行通信的操作。

1. 总体设计



**关于三个不同版本的设计**

在版本一中，将消费者和商家共同继承用户基类，在商家类下面完成商家的管理商品、商品打折等专属于商家的操作。食物、书本、衣物等共同继承商品基类。在平台类中，可以对用户和商家进行实例化操作，将所有用户与商品信息都存放在平台类中，在平台类中，可以完成不需要登录的操作，如筛选和查看商品等。print类属于每一个有输出的类中，这样可以调用其的输入合法化的模板。在main函数中，实例化平台类，根据不同情况进行平台下的不同操作。

在版本二中，需要增加一个购物车类，用来表示购物商品和购买量，在每个消费者中存放购物中的商品和订单中的商品。并且在消费者类下面完成消费者的操作，如添加、修改购物车，生成或取消订单。在平台下面，因为所有的商品和商家信息存在平台下，因此关于展示购物车和订单等，因为与不同商家中的打折情况相关，这些操作放在平台类下面执行，包括计算与支付订单。

在版本三中，只需要增加一个服务端，用来建立连接和接收与发送信息即可，同时，要把服务端中所有的输出转变为字符流的输入，缓存起来，在适当的时机发送和接收消息，其余处理皆不用变化。在客户端，仅需要向服务端传送消息和接收消息即可。

三、数据结构说明

**1、版本1**

account.h

|  |
| --- |
| class Account  {  protected:  int number; //记录用户的编号  std::string id; //记录用户的名称  std::string passwd; //记录密码  double balance; //记录余额  public:  User userType;  public:  Account(const std::string& id\_="", const std::string& passwd\_="",const double& balance\_=0,const User & userType\_= CONSUMER);  Account(const Account& acc);  ~Account();  int getNumber() const { return number; }  void setNumber(const int& number\_) { number = number\_; }  std::string getId() const { return id; }  void setId(const std::string& i) { id = i;}  std::string getPasswd() const { return passwd; }  void setPasswd(const std::string& passw) { passwd = passw; }  double getBalance() const { return balance; }  void setBalance(const double& balanc) { balance = balanc; }  virtual User getUserType()=0;  //bool operator==(const std::string& id\_);  void changePasswd(); //修改用户密码  void manageBalance(); //管理用户  void queryBalance()const; //查询余额  void addBalance(); //充值  void consumeBalance(); //消费  }; |

consumer.h

|  |
| --- |
| class Consumer :  public Account  {  public:  Consumer(const int number\_, const std::string& id\_="", const std::string& passwd\_="", const double& balance\_=0, const User& userType\_=CONSUMER);  Consumer(const Consumer& acc);  ~Consumer();  public:  virtual User getUserType() override;    }; |

merchant.h

|  |
| --- |
| class Merchant :  public Account  {  public:  std::vector<Goods\*> merGoods; //存储该商家管理的商品  std::map<int, double> merDiscount; //存储该商家都不同类商品的打折情况  public:  Merchant(const int number\_,const std::string& id\_ = "", const std::string& passwd\_ = "", const double& balance\_ = 0, const User& userType\_ = MERCHANT);  Merchant(Merchant& acc);  ~Merchant();  public:  virtual User getUserType() override ; //得到用户类型  void addgoods(std::string filename, std::vector<Goods\*>& allGoods); //增加商品  void manageGoods(std::vector<Goods\*> & allGoods); //管理商品  void haveDiscount() ; //对同类商品进行打折  }; |

goods.h

|  |
| --- |
| class Goods  {  protected:  int number; //表示商品所属的商家编号  std::string name; //商品名称  std::string descrip; //商品描述  double price; //商品价格  int amount; //商品剩余量  Good goodType; //商品类型  public:  Goods(const std::string& name\_="", const std::string& descrip\_ = "", const double& price\_ = 0, const int& amount\_ = 0, const Good& goodType\_=FOOD);  Goods(const Goods& goods);  ~Goods();  //以下为改变和获得成员变量的函数  virtual double getPrice(double discount) { return price \* discount; }  int getNumber() const { return number; }  void setNumber(const int& id) { this->number = number; }  std::string getName() const { return name; }  void setName(const std::string& name) { this->name = name; }  std::string getDescrip() const { return descrip; }  void setDescrip(const std::string& descri) { descrip = descri; }  double getPrice() const { return price; }  void setPrice(const double& pric) { price = pric; }  int getAmount() const { return amount; }  void setAmount(const int& amoun) { amount = amoun; }  Good getgoodType() const { return goodType; }  void setgoodType(const Good &goodTYp) { goodType = goodTYp; }  }; |

book.h

|  |
| --- |
| class Book :  public Goods  {  public:  Book(const int number\_,const std::string& name\_, const std::string& descrip\_ = "", const double& price\_ = 0, const int& amount\_ = 0, const Good& goodType\_ = FOOD);  Book(const Book& goods);  ~Book();  virtual double getPrice(double discount = 1) override;  }; |

clothes.h

|  |
| --- |
| class Clothes :  public Goods  {  public:  Clothes(const int number\_,const std::string& name\_, const std::string& descrip\_ = "", const double& price\_ = 0, const int& amount\_ = 0, const Good& goodType\_ = FOOD);  Clothes(const Clothes& goods);  ~Clothes();  virtual double getPrice(double discount = 1) override;  }; |

food.h

|  |
| --- |
| class Food :  public Goods  {  public:  Food(const int number\_ ,const std::string& name\_, const std::string& descrip\_ = "", const double& price\_ = 0, const int& amount\_ = 0, const Good& goodType\_ = FOOD);  Food(const Food& goods);  ~Food();  virtual double getPrice(double discount = 1) override;  }; |

print.h

|  |
| --- |
| class print  {  public:  void printfi() const;  void printfm() const;  void printfc() const;  template<class T>  void input(T& x) const {  std::cin >> x;  //确保读入的类型正确且最后以\n收尾不存在多余的字符  while (std::cin.get() != '\n' || std::cin.fail()) {  std::cout << "请输入正确的输入：" << std::endl;  std::cin.clear(); //清空错误码  std::cin.ignore(std::numeric\_limits< std::streamsize >::max(), '\n');//忽略最后多余的字符  std::cin >> x;  }  }  }; |

platform.h

|  |
| --- |
| class Platform  {  public:  std::vector<Account\*> allAccount; //记录所有商家信息  std::vector<Goods\*> allGoods; //记录所有商品信息  //std::map<int, double> allDiscount;  private:  int number = 0;  public:  Platform(); //构造函数  ~Platform(); //析构函数  void showGoodsInfo(const std::vector<Goods\*>& merGoods) const; //展示商家商品信息  void searchGoods() const; //筛选所有商品  void registerUser(); //注册用户  Account\* loginUser(); //登录用户  bool initAccount(); //初始化账户  bool initGoods(); //初始化商品  void upgradeAccount() const; //在文件中更新用户信息  void upgradeGoods() const; //在文件中更新商品信息  }; |

**2、版本2新增**

shopcard.h

|  |
| --- |
| class Shopcard  {  Goods\* goodsCard;  int buyNumber=0;  public:  Shopcard(Goods\* & good\_,const int & buyNumber\_);  Shopcard(const Shopcard & shop\_);  ~Shopcard();    int getbuyNumber() const { return buyNumber; }  void setbuyNumber(const int& number) { this->buyNumber = number; }    Goods\* getGoodsCard() const { return goodsCard; }  void setGoodsCard(Goods\*& goodsCar) { goodsCard = goodsCar; }    }; |

consumer.h

|  |
| --- |
| class Consumer :  public Account  {  public:  std::vector <Shopcard\* > shopcart;  std::vector <Shopcard\* > ordercart;  //std::vector <std::vector <Shopcard\* > > orders;  public:  Consumer(const int number\_, const std::string& id\_="", const std::string& passwd\_="", const double& balance\_=0, const User& userType\_=CONSUMER);  Consumer(const Consumer& acc);  ~Consumer();  public:  virtual User getUserType() override;  //添加购物车  void addShopcart(std::vector<Goods\*>& allGoods);  //修改购物车商品  void modifyShopcart( );  //删除购物车中的商品  void deleteShopcart();  //生成订单  void generateOrder();  //取消订单  void cancelOrder();  }; |

platform.h

|  |
| --- |
| class Platform  {  public:  std::vector<Account\*> allAccount; //记录所有商家信息  std::vector<Goods\*> allGoods; //记录所有商品信息  //std::map<int, double> allDiscount;  private:  int number = 0;  public:  Platform(); //构造函数  ~Platform(); //析构函数  void showGoodsInfo(const std::vector<Goods\*>& merGoods) const; //展示商家商品信息  void searchGoods() const; //筛选所有商品  void registerUser(); //注册用户  Account\* loginUser(); //登录用户  void initAccount(); //初始化账户  void initGoods(); //初始化商品  void upgradeAccount() const; //在文件中更新用户信息  void upgradeGoods() const; //在文件中更新商品信息  void showShopcart(Consumer\* & consumer ) const; //展示购物车  void showOrder(Consumer\* & consumer) const; //展示订单  double calOrder(Consumer\*& consumer) const; //计算订单  void payloadOrder(Consumer\*& consumer, double& sum);  }; |

print.h

|  |
| --- |
| class print  {  public:  void printfi() const;  void printfm() const;  void printfc() const;  void printfcart() const;  void printforder() const;  template<class T>  void input(T& x) const {  std::cin >> x;  //确保读入的类型正确且最后以\n收尾不存在多余的字符  while (std::cin.get() != '\n' || std::cin.fail()) {  std::cout << "请输入正确的输入：" << std::endl;  std::cin.clear(); //清空错误码  std::cin.ignore(std::numeric\_limits< std::streamsize >::max(), '\n');//忽略最后多余的字符  std::cin >> x;  }  }  }; |

**3、版本3新增**

服务端

serve.h

|  |
| --- |
| class serve  {  //定义服务端套接字，接受请求套接字  SOCKET s\_server;  SOCKET s\_accept;  //服务端地址客户端地址  SOCKADDR\_IN server\_addr;  SOCKADDR\_IN accept\_addr;  std::stringstream send\_buffer;  std::stringstream receive\_buffer;  //临时缓存  char tempdata[300];  int ifsend = 1;  int ifreceive = 1;  public:  void initialization();//初始化  void setup();//建立连接  //void sendBuffer(const std::string & s);  void close();//关闭连接  void mySend();//自定义发送  bool iflegal();//判断连接是否合法建立  std::stringstream& myReceive();//自定义收取  std::stringstream& out() { return receive\_buffer; }//输出流  std::stringstream& in() { return send\_buffer; }//输入流  };  extern serve ser; |

客户端

main.cpp

|  |
| --- |
| int main() {  //定义长度变量  int send\_len = 0;  int recv\_len = 0;  //定义发送缓冲区和接受缓冲区  //char send\_buf[1000];  char recv\_buf[1000];  std::string send\_buffer;  //std::string recv\_buffer;  //定义服务端套接字，接受请求套接字  SOCKET s\_server;  //服务端地址客户端地址  SOCKADDR\_IN server\_addr;  //初始化套接字库  WORD w\_req = MAKEWORD(2, 2);//版本号  WSADATA wsadata;  int err;  err = WSAStartup(w\_req, &wsadata);  if (err != 0) {  std::cout << "初始化套接字库失败！" << std::endl;  }  else {  std::cout << "初始化套接字库成功！" << std::endl;  }  //检测版本号  if (LOBYTE(wsadata.wVersion) != 2 || HIBYTE(wsadata.wHighVersion) != 2) {  std::cout << "套接字库版本号不符！" << std::endl;  WSACleanup();  }  else {  std::cout << "套接字库版本正确！" << std::endl;  }  //填充服务端信息  server\_addr.sin\_family = AF\_INET;  server\_addr.sin\_addr.S\_un.S\_addr = inet\_addr("127.0.0.1");  server\_addr.sin\_port = htons(8080);  //创建套接字  s\_server = socket(AF\_INET, SOCK\_STREAM, 0);  if (connect(s\_server, (SOCKADDR\*)&server\_addr, sizeof(SOCKADDR)) == SOCKET\_ERROR) {  std::cout << "服务器连接失败！" << std::endl;  WSACleanup();  }  else {  std::cout << "服务器连接成功！" << std::endl;  }  //发送,接收数据  while (1) {    recv\_len = recv(s\_server, recv\_buf, 1000, 0);  if (recv\_len < 0) {  std::cout << "接受失败！" << std::endl;  break;  }  else {  //std::cout << "服务端信息:\n" << recv\_buf << std::endl;  std::cout << recv\_buf;  }  std::cout << "请输入发送信息:";  std::cin >> send\_buffer;  //每次输入后都要加上一个\n表示字符串的结束  send\_buffer += "\n";  send\_len = send(s\_server, send\_buffer.c\_str(), sizeof(send\_buffer), 0);  if (send\_len < 0) {  std::cout << "发送失败！" << std::endl;  break;  }  }  //关闭套接字  closesocket(s\_server);  //释放DLL资源  WSACleanup();  return 0;  } |

1. 模块说明

**1、版本1**

·Acount类

此为用户基类，在该用户中含有用户编号、用户名称、密码、余额等信息。并且有纯虚函数getUserType(),因此该类不能实例化。在该类中完成修改密码，查询、充值、消费余额等功能。

·Consumer类

此类公有继承了用户类，实例化表示一个用户。

·Merchant类

此类公有继承了用户类，实例化表示一个商家。在该类中，存储着该商家管理商品的指针数组，并且用了map格式存储了该商家对不同类商品的打折信息。并且商家完成的功能：添加商品、管理商品、打折等功能在该类中实现。

·Good类

此为商品基类，在该商品中含有商品所属商家编号、名称、描述、价格、剩余量、类型等信息。并含有虚函数getPrice。

·Book类/Clothes类/Food类

这三个类都公有继承了商品类并重写了getPrice的虚函数。

·Print类

在该类中含有初始情况、商家情况、消费者情况等提示消息。并且含有正确输入的模板。

·Platform类

在平台类中，储存了所有商家和商品类的指针。在该平台类下，完成展示商品信息，筛选信息，注册账户，登录账户等功能。并且还需要在平台运行时初始化账户与商品信息，同时在结束时要更新账户与商品信息。

**2、版本2新增**

·Shopcard类

此类为购物车类，包含了商品的指针和购买量。每购买一个商品则对应一个购物车类。

·Consumer类

在该类中有购物车与订单的vector，其类型都为购物车类的指针，存放着在购物车中的商品和在订单中的商品。在该类中实现消费者的添加购物车，修改购物车中的商品，删除购物车中的商品，生成订单，取消订单等功能。

·Platform类

在该类中新增了展示购物车、展示订单、计算订单金额、支付订单等功能。

·Print类

在该类中增加了购物车管理和订单管理的提示输出功能。

**3、版本3新增**

·Serve类

在该类中实现服务端的连接。定义套接字、接收发送地址、输入输出流。定义初始化变量、建立连接、关闭连接等操作。并自定义发送与接收，输入流与输出流。此外在该类中判断连接是否正常运行。并且该类的实例化采用了全局的方式，这样文件下所有的缓存都会缓存到serve这个类中。

·客户端的main

在客户端的main中，建立套接字，实现发送与接收的操作。本客户端只完成接收数据与发送数据，其余处理都放在了服务端。

五、经验与教训

1、对于**继承与派生**的认识更加深刻，对于继承同一个类的不同类别怎么合适存储？

在此实验中，最主要的是怎么合适的存储用户和商家的信息。因为用户有商家和消费者两种类别，且商家和消费者的权限与功能是不同的，但是在创建用户时，如果vector中存放的是消费者或者用户类，那个vector的类型就会不确定，如果将商家和消费者分别存在不同的vector中，那么操作两个存储用户的vector就会不太方便。最后的解决方法是，创建一个存储基类account\*指针的vector，在创建用户时，创建指向消费者或者商家的指针，将指针存到vector中，这时候存储的指针类型就变为指向基类的指针，但事实上，其指针创建时开辟了该类型大小的区域。因此，如果需要取出，则需要强制转化为商家或消费者，这时候该指针可以正常的完成商家或消费者的有关操作。对于存储不同商品类别的信息也是采取上述的同样操作即可。

该设计使我对c++中类的指针、类的实例、不同类之间的相互转化、基类和派生类等的关系都有了更深层次的认识。并且通过指针，我们可以在不同类中对同一片区域进行操作，使操作更加便捷，避免了重复的内存拷贝。

2、关于在指针数组中怎么找到对应**匹配的值**？

在该实验中，有一项功能经常被用到，就是需要查找该商品或用户名称是否已经存在，来判断该步骤的操作是否合法，如何查找就是需要考虑的问题。自然，我们可以通过遍历指针vector来搜索是否有满足的指针，但是在写的时候，尝试用了find\_if来简化多行的运算，并且还能减少出错。在用了find\_if时，使用了lambda表达式，例子如下：

auto it = std::find\_if(merGoods.begin(), merGoods.end(), [&](Goods\* A)-> bool {return A->getName() == name\_;}

因为在指针数组中，如果单纯用find无法找到其值，因为find只能找到元素，而我们并不知道该名称对应的指针地址，只知道该指针指向的类的对应名称，因此这时候需要用find\_if函数，在条件判断用了lambda式子，即判断对于某指针指向类的名称与所给名称相同，则表示找到了，否则没有找到。

3、关于**输入合法化**如何方便处理？

在该实验中涉及到很多输入的情形，如何方便快捷的判断输入是否合法，在此实验中，采用了模板的方式进行简化，每次输入时调用函数进行输入，这样可以方便快捷的判断输入是否符合类型，与课上学习到的内容刚好有了连接。

4、关于**const**的规范使用

在该实验中明确了const的实验规范，在不需要修改该类的时候，添加const来说明此函数不修改类。当然，若在函数中修改了类中内容并添加了const函数，此时就会发生错误。此外，尤其要注意指针的操作，若返回了const指针，那么后续的指针操作中就会发生类型不匹配的错误，虽然没有修改，但是这样也会发生错误，需要我们注意。

5、如何运用**字符流存储通信字符缓存**？

在socket编程中，两边通信的字符串可能需要有字符串缓存之后再进行通信，这时候就需要把要发送的字符串进行缓存。在该实验中，采用了stringstream的方式，通过字符流的方式将要发送的数据缓存下来。通过这种方式实现了字符串的拼接，并且对于程序的读取结构不会有较大的影响。

通过字符流的方式，对于输入输出符不用有太大的改变。只需要写上自己的输入输出函数，不断的从输入输出流读入读出即可。需要注意的是，在客户端读入字符串时，在发送前需要加上‘\n’，因为读取时是不读入换行符的，在发送时需要加上。

对于某一方突然关闭的异常情形，有一个判断是否合法的函数，需要在读入时进行检验，否则会出现不断读取错误无法退出的情况。可以采用抛出异常的方式，当捕获到连接发生异常时则抛出错误，断开连接。