



A C++ Console Application

ONLINE SHOPPING PLATFORM

BY: Gayanthika Shankar



Project Overview

- This is a shopping program where you can:
 - Look at different products to buy
 - Add things you like to your cart
 - Buy the items in your cart
 - See what you bought before
 - It's like a real online store but simpler!
-

Main Features

- Browse and see different types of products
- Add products to your shopping cart
- Remove items you don't want anymore
- Check out and place orders
- View your order history

```
Welcome to the Online Shopping System
Please enter your name: Shopper1
Hello, Shopper1, let's start shopping.

----- Online shopping SystemM-----
1) Browse Products
2) View Cart
3) Remove Item from Cart
4) Checkout
5) View Order History
6) Exit
-----
Enter your choice: 1
```


Program Flow

- Program starts and welcomes user
 - User enters name
 - Main menu appears with options
 - User can browse products and add to cart
 - User checks out when ready
 - Order is processed and recorded
 - User can continue shopping or exit
-

Classes and Objects

- Classes are blueprints, Objects are the actual things:
- Product class: Blueprint for all items (abstract)
- Electronics, Clothing, Book: Specific product types
- ShoppingCart: Holds selected items
- User: Stores customer information and cart
- OrderProcessor: Handles order processing

Classes and Objects

```
//base class
class Product {
protected:
    std::string name;
    double price;
    static int totalProducts;
```

```
//derived class
class Electronics : public Product {
private:
    std::string brand;
    int warrantyMonths;
```

```
class User {
private:
    std::string username;
    ShoppingCart cart;
    std::stack<std::shared_ptr<Product> > orderHistory;
```

code lines 13, 59, 78, 96, 115, 175, 236

Constructors

- What are constructors? Special functions that create new objects

Where implemented: Lines 22, 66, 84, 102, 121, 182

Constructors

```
public:
    //constructor
    Product(std::string n, double p) {
        this->name = n;
        this->price = p;
        totalProducts++;
    }

    //virt destructor
    virtual ~Product() {
        totalProducts--;
    }
```

```
public:
    Clothing(std::string n, double p, std::string s, std::string m)
        : Product(n, p) {
        this->size = s;
        this->material = m;
    }
```

```
✓ public:
    ShoppingCart() {} //constructor
```

Where implemented: Lines 22, 66, 84, 102, 121, 182

Inheritance

- When a child class gets features from a parent class

In our code:

- Product (parent)
- Electronics, Clothing, Book (children)

```
Product (Parent Class)
├── Electronics (line 71)
├── Clothing (line 87)
└── Book (line 103)
```

Why important: Reuses code and organizes things logically

Polymorphism

- Different objects respond to the same function in their own way

In our code:

- Each product type displays details differently
- Base class: `virtual void displayDetails() const = 0;`
- Electronics: Shows brand and warranty
- Clothing: Shows size and material
- Book: Shows author and pages

Why important: We can treat all products the same but get customized behavior

Polymorphism

```
//virt destructor  
virtual ~Product() {  
|   totalProducts--;  
}
```

```
std::stack<std::shared_ptr<Product>> orderHistory;
```

```
virtual void displayDetails() const = 0;
```

```
void displayDetails() const /* override */ {  
|   std::cout << "Electronics: " << name << ", Brand: " << brand  
|   |   |   << ", Price: $" << price << ", Warranty: " << warrantyMonths << " months" << std::endl;  
}
```

where implemented: lines 29, 39, 72, 90, 108, 131, 179, etc

Friend Functions and Classes

- Friend Function: Can access private data of a class
 - applyDiscount function
 - Allows changing price directly
- Friend Class: One class can access private parts of another
 - User class is a friend of ShoppingCart
 - Allows User to directly access ShoppingCart's items

Why important: Controlled access to normally private data

Friend Functions and Classes

```
friend void applyDiscount(Product& product, double percentage);
```

```
void applyDiscount(Product& product, double percentage) {  
|   product.price -= (product.price * percentage / 100);  
}
```

```
friend class User;
```

```
bool placeOrder() {  
|   if (cart.items.empty()) {
```

seen in lines: 50, 55, 172, 193, 214

Operator Overloading

- What is it? Making operators like + or > work with our custom objects
- In our code: Compare product prices using > and <

```
bool operator>(const Product& other) const {  
|   return price > other.price;  
}  
  
bool operator<(const Product& other) const {  
|   return price < other.price;  
}
```

Why important: Makes code more readable and intuitive

Where implemented: Lines 42-48

Static Members

- What are static members? Variables that belong to the class, not individual objects

Why important: Tracks information about all products together

Static Members

```
static int totalProducts;
```

```
totalProducts++;
```

```
int Product::totalProducts = 0;
```

```
virtual ~Product() {  
|   totalProducts--;  
}
```

```
static int getTotalProducts() { return totalProducts; }
```

lines: 18, 53, 37, 25, 30

Memory Allocation & Pointers

- Dynamic Memory: Created during program runtime
- `new Electronics(...)`
- Raw Pointers: Direct memory references
- `User* user` in `OrderProcessor`
- Smart Pointers: Self-cleaning pointers
- `std::shared_ptr<Product>`

Why important: Allows creating objects when needed and manages memory safely

Memory Allocation & Pointers

```
inventory.push_back(std::shared_ptr<Electronics>(new Electronics("Laptop", 999.99, "TechBrand", 24)));  
inventory.push_back(std::shared_ptr<Electronics>(new Electronics("Smartphone", 699.99, "PhoneCo", 12)));  
inventory.push_back(std::shared_ptr<Electronics>(new Electronics("Headphones", 149.99, "AudioTech", 6)));  
inventory.push_back(std::shared_ptr<Clothing>(new Clothing("T-Shirt", 19.99, "M", "Cotton")));  
inventory.push_back(std::shared_ptr<Clothing>(new Clothing("Jeans", 49.99, "32", "Denim")));  
inventory.push_back(std::shared_ptr<Clothing>(new Clothing("Hoodie", 39.99, "L", "Fleece")));  
inventory.push_back(std::shared_ptr<Book>(new Book("Programming C++", 39.99, "Albert E. ", 450)));  
inventory.push_back(std::shared_ptr<Book>(new Book("Data Structures and Algorithms", 29.99, "Martha Williams", 380)));
```

```
std::queue<User*> orderQueue; //processing order
```

```
void addOrder(User* user) {
```

```
User* user = orderQueue.front();
```

```
processor.addOrder(&user);
```


Access Modifiers

- Public: Anyone can use these (like product names)
- Private: Only the class itself can use these
- Protected: Only the class and its children can use

```
class Product {  
protected:  
    std::string name;  
    double price;  
    static int totalProducts;  
};
```

```
class Clothing : public Product {  
private:  
    std::string size;  
    std::string material;  
public:  
    Clothing(std::string n, double p, std::string s, std::string m)  
        : Product(n, p) {  
        this->size = s;  
        this->material = m;  
    }  
    void displayDetails() const {  
        std::cout << "Clothing: " << name << ", Size: " << size  
        << ", Material: " << material << ", Price: $" << price << std::endl;  
    }  
};
```

Why important: Protects data from accidental changes

Passing by Value/Reference

- Pass by Value: Makes a copy
- Pass by Reference: Uses the original

```
friend void applyDiscount(Product& product, double percentage);
```

```
Product(std::string n, double p) {
```

```
ShoppingCart& getCart() { return cart; }
```

```
//price comparison  
bool operator>(const Product& other) const {
```

```
friend void applyDiscount(Product& product, double percentage);
```

Why important: References avoid copying large objects and allow changing the original

lines = 50, 189, 42, 22, 50 etc

Data Structures

- Vector: Stores collections that can grow/shrink
- Stack: Last-in, first-out storage
- Queue: First-in, first-out storage

Why important: Organizes data in useful ways for different purposes

Data Structures

```
std::vector<std::shared_ptr<Product> > items;
```

```
std::queue<User*> orderQueue;
```

```
items.push_back(product);
```

```
std::stack<std::shared_ptr<Product> > orderHistory;
```

```
User* user = orderQueue.front();
```

```
orderHistory.top()->displayDetails();
```

lines: 118, 123, 179, 231, 238, 252 etc

Getters and Setters

- Getters: Return private data safely
- Setters: Change private data safely

```
std::string getName() const { return name; }  
double getPrice() const { return price; }  
void setPrice(double p) { price = p; }  
  
static int getTotalProducts() { return totalProducts; }
```

Where implemented: Lines 33-37, 35, 186, 189, 164, 262,

Why important: Controls access to private data

References

Aliases for existing variables

```
friend void applyDiscount(Product& product, double percentage);
```

```
void displayProductCatalog(const std::vector<std::shared_ptr<Product> >& inventory) {}
```

```
ShoppingCart& getCart() { return cart; }
```

```
bool operator>(const Product& other) const {
```

```
bool operator<(const Product& other) const {
```

Where implemented: lines 50, 285, 189, 42, 46

Why important: Makes code safer and more efficient

Smart Pointers

Pointers that manage their own memory

Benefits:

- Automatically deletes memory when no longer needed
 - Multiple parts of code can share ownership
 - No memory leaks!
-

Smart Pointers

```
std::vector<std::shared_ptr<Product> > items;
```

```
items.push_back(product);
```

```
user.getCart().addProduct(inventory[productChoice-1]);
```

```
void addProduct(std::shared_ptr<Product> product)
```

```
inventory.push_back(std::shared_ptr<Electronics>(new Electronics("Laptop", 999.99, "TechBrand", 24)));  
inventory.push_back(std::shared_ptr<Electronics>(new Electronics("Smartphone", 699.99, "PhoneCo", 12)));  
inventory.push_back(std::shared_ptr<Electronics>(new Electronics("Headphones", 149.99, "AudioTech", 6)));  
inventory.push_back(std::shared_ptr<Clothing>(new Clothing("T-Shirt", 19.99, "M", "Cotton")));  
inventory.push_back(std::shared_ptr<Clothing>(new Clothing("Jeans", 49.99, "32", "Denim")));  
inventory.push_back(std::shared_ptr<Clothing>(new Clothing("Hoodie", 39.99, "L", "Fleece")));  
inventory.push_back(std::shared_ptr<Book>(new Book("Programming C++", 39.99, "Albert E. ", 450)));  
inventory.push_back(std::shared_ptr<Book>(new Book("Data Structures and Algorithms", 29.99, "Martha Williams", 380)));
```

lines implemented: 118, 179, 299, 122, 302-09, 123, 326, etc

Thank you!
