2019 Exam Memo

-Rahul Nundlall

Question 1

1.1

(a) **Rely on comments to explain how a function is implementing its task.**
False. While comments can be useful, relying on them for understanding how a function works is not considered good practice. The code itself should be self-explanatory through meaningful variable and function names and a well-organized structure.

(b) **Name types using nouns, and functions using verbs.**
True. This is a well-accepted naming convention in many programming languages. Naming types with nouns and functions with verbs makes the code more readable and self-explanatory.

(c) **Describe the return value type within the function name so that programmers have a better understanding of the function.**
False. While it's crucial to understand a function's return type, this information is usually conveyed in the function signature and documentation, rather than in the function name itself

(d) **Use abbreviations in variable and function names.**
False. Abbreviations can lead to confusion and decrease readability. It's generally advised to use full words to make the code as self-explanatory as possible.

(e) **Phrase functions that return a Boolean value as questions.**
True. This is a common naming convention for Boolean functions. For example, **isEmpty()**, **isAvailable()**, etc., make it clear that the function will return either **true** or **false**.

1.2

Deals with the transform function, so it may be prudent to learn about it. The **std::transform** function applies a given function to a range and stores the output in another range. Here's the basic syntax:

```
// Unary operation
std::transform(InputIt first1, InputIt last1, OutputIt d_first, UnaryOperation unary_op);

// Binary operation
std::transform(InputIt first1, InputIt last1, InputIt first2, OutputIt d_first, BinaryOperation binary_op);
```

Example of a unary operation:

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
   std::vector<int> vec{1, 2, 3, 4, 5};
   std::vector<int> result(vec.size());

   std::transform(vec.begin(), vec.end(), result.begin(), [](int i) { return i * i; });

   for (const auto& n : result) {
      std::cout << n << " ";
   }

   return 0;
}
```

Example of binary operation:

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
   std::vector<int> vec1{1, 2, 3};
   std::vector<int> vec2{4, 5, 6};
   std::vector<int> result(vec1.size());

   std::transform(vec1.begin(), vec1.end(), vec2.begin(), result.begin(), std::plus<int>());

   for (const auto& n : result) {
      std::cout << n << " ";
   }

   return 0;
}
```

Other Key Functions in <algorithm>

1. **std::sort**: Sorts the elements in a range.

```
std::sort(vec.begin(), vec.end());
```

2. **std::find**: Finds the first element satisfying specific criteria.

```
auto it = std::find(vec.begin(), vec.end(), 3);
```

3. **std::count**: Counts elements satisfying specific criteria.

```
int n = std::count(vec.begin(), vec.end(), 3);
```

4. **std::copy**: Copies elements from one range to another.
std::copy(src.begin(), src.end(), dest.begin());

5. **std::reverse**: Reverses a range.
std::reverse(vec.begin(), vec.end());

6. **std::accumulate**: Accumulates values in a range.
int sum = std::accumulate(vec.begin(), vec.end(), 0);

1.2 Given the following program, which of the statements concerning it are true?

(5 marks)

```
1 int calculate_length(const string& word) { return word.length(); }
2
3 int main()
4 {
5    auto words = vector<string>{"This", "is", "a", "short", "sentence"};
6    auto word_lengths = vector<int>(words.size()); // for results
7    transform(begin(words), end(words), begin(word_lengths),
         calculate_length);
8    return 0;
9 }
```

(a) An in-place transformation can be used instead of storing the results in `word_lengths`.

(b) The `transform` function does not call the `calculate_length` function when the iterator moving over the range is equal to: `end(words)`.

(c) The following line of code is equivalent to that on line 6:
`auto word_lengths = vector<int>{words.size()};`

(d) `calculate_length` is known as a *function object*.

(e) The implementation of the `transform` function does not use `vector<T>::push_back`.

(a) An in-place transformation can be used instead of storing the results in word_lengths.
False. The **transform** function requires an output iterator where the transformed elements will be stored. The **words** vector contains strings, and we are calculating their lengths, which are integers. Due to the type mismatch, in-place transformation isn't feasible here.

(b) **The transform function does not call the calculate_length function when the iterator moving over the range is equal to: end(words).**
True. The **transform** function operates in the range **[begin(words), end(words))**, where **end(words)** is not included. So **calculate_length** will not be called when the iterator equals **end(words)**.

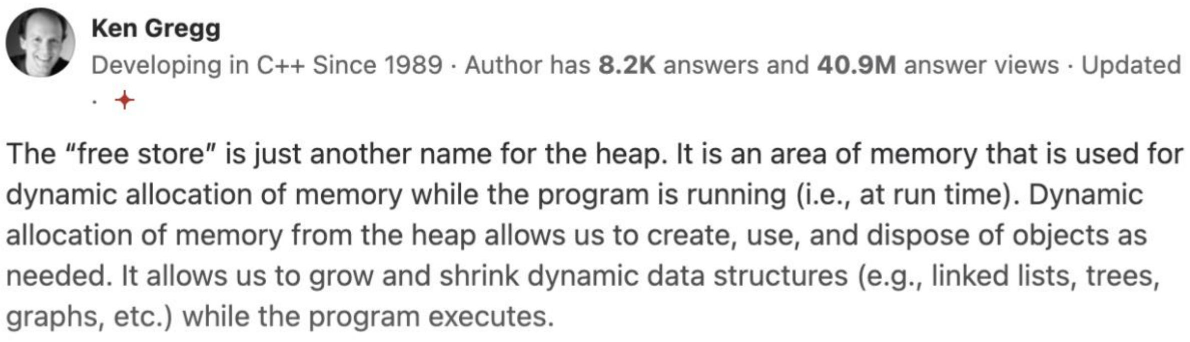
(c) **The following line of code is equivalent to that on line 6: auto word_lengths = vector<int>{words.size()};**
False. The code **auto word_lengths = vector<int>{words.size()};** would create a vector with a single element whose value is the size of **words**. The original line **auto word_lengths =**

vector<int>(words.size()); creates a vector of integers with a size equal to **words.size()**, all initialized to zero.
(d) **calculate_length is known as a function object.**
False. calculate_length is a regular function, not a function object (also known as a functor). A function object is an instance of a class that defines the operator **()**.

(e) **The implementation of the transform function does not use vector<T>::push_back.**
True. The **transform** algorithm typically assigns to the elements in the destination range and doesn't need to change the size of the output container. Therefore, it would not generally use **vector<T>::push_back**.

1.3

(a) **The terms free store and heap are equivalent.**
True. In the context of C++ memory management, the terms "free store" and "heap" are often used interchangeably to describe the area of memory from which dynamic memory allocations are made


Ken Gregg
Developing in C++ Since 1989 · Author has **8.2K** answers and **40.9M** answer views · Updated ·

The "free store" is just another name for the heap. It is an area of memory that is used for dynamic allocation of memory while the program is running (i.e., at run time). Dynamic allocation of memory from the heap allows us to create, use, and dispose of objects as needed. It allows us to grow and shrink dynamic data structures (e.g., linked lists, trees, graphs, etc.) while the program executes.

(b) **Smart pointers are intended to be created on the stack and not on the heap.**
True. Smart pointers like **std::unique_ptr** and **std::shared_ptr** are generally intended to be allocated on the stack. They manage the memory of objects on the heap internally, making manual memory management easier.

(c) **Calling delete on a pointer only deletes the pointer, not what is it pointing to.**
False. The **delete** operator frees the memory that the pointer is pointing to and calls the destructor of the object. It does not delete the pointer itself; the pointer becomes a dangling pointer.

(d) **It is safe to call delete multiple times on a pointer without any side effects, if the pointer is set to nullptr.**
True. Calling **delete** on a **nullptr** is a no-op (no operation). However, if you don't set the pointer to **nullptr** after the first **delete**, calling **delete** again would result in undefined behavior.

(e) **A smart pointer cannot prevent memory leaks if an exception is thrown.**
False. One of the advantages of smart pointers is that they automatically manage the memory, including during exceptions. When a smart pointer goes out of scope (e.g., due to

an exception), its destructor will be called, and it will free the memory it owns, preventing a memory leak.

1.4

(a) **When setting up a C++ project which makes use of SFML there is no need to add the SFML header files to the project in the IDE, if the include path to the header files has been specified in the project settings.**
True. If the include path has been set in the project settings, the compiler will know where to look for the SFML headers. Therefore, you won't need to add them manually to your project in the IDE.

(b) **SFML provides inheritance hierarchies which can be extended (derived from) for your own application.**
True. SFML has object-oriented designs for many of its components (e.g., **sf::Drawable**, **sf::Transformable**), and you can extend these through inheritance to create your own customized classes.

(c) **The functionality provided by SFML is part of the std namespace.**
False. SFML has its own namespace (**sf**), separate from the standard (**std**) namespace.

(d) **Linking with the SFML library files is done after the code is compiled.**
True. Linking is generally the final phase in generating an executable and it happens after the code has been compiled. You'd link against the compiled SFML libraries to produce the final executable.

(e) **Dynamically linking to the SFML libraries means that the size of the executable using SFML is larger than if it were statically linked to the libraries.**
False. Generally, dynamically linking results in a smaller executable size, because the code for the libraries is not included in the executable. Instead, the libraries are loaded at runtime. In static linking, the library code gets incorporated into the executable, making it larger.

Question 2

(a) We can follow two approaches, one where we initialize the default value in the constructor and another where we create an overloaded instance of the constructors, which will in turn call the no parameter constructor for library item, thereby completing the task. However given the phrasing of the question, it is more likely that the first approach would be correct.

Approach 1:
```
class LibraryItem {
public:
   LibraryItem(int total_copies = 1) : total_copies_{total_copies}, copies_on_loan_{0} {}
   // ... (rest remains the same)
private:
   int total_copies_;
```

```
    int copies_on_loan_;
};

class Book : public LibraryItem {
public:
    Book(const string& title, const string& author, int copies = 1)
        : LibraryItem{copies}, title_{title}, author_{author} {}
    // ... (rest remains the same)
private:
    string title_;
    string author_;
};

class DVD : public LibraryItem {
public:
    DVD(const string& title, int runtime, int copies = 1)
        : LibraryItem{copies}, title_{title}, runtime_{runtime} {}
    // ... (rest remains the same)
private:
    string title_;
    int runtime_;
};
```

Approach 2:

```
class LibraryItem {
public:
    LibraryItem(int total_copies) : total_copies_{total_copies}, copies_on_loan_{0} {}
    LibraryItem() : total_copies_{1}, copies_on_loan_{0} {} // new constructor
    // ... (rest remains the same)
private:
    int total_copies_;
    int copies_on_loan_;
};

class Book : public LibraryItem {
public:
    Book(const string& title, const string& author, int copies)
        : LibraryItem{copies}, title_{title}, author_{author} {}
    Book(const string& title, const string& author)
        : LibraryItem{}, title_{title}, author_{author} {}  // new constructor
    // ... (rest remains the same)
private:
    string title_;
    string author_;
};

class DVD : public LibraryItem {
public:
    DVD(const string& title, int runtime, int copies)
        : LibraryItem{copies}, title_{title}, runtime_{runtime} {}
    DVD(const string& title, int runtime)
        : LibraryItem{}, title_{title}, runtime_{runtime} {} // new constructor
    // ... (rest remains the same)
private:
    string title_;
    int runtime_;
};
```

(b)

```
class LibraryItem {
public:
    static int total_items_on_loan;  // Static member to keep track of total items on loan

    LibraryItem(int total_copies) : total_copies_{total_copies}, copies_on_loan_{0} {}

    void returnItem() {
        copies_on_loan_--;
        total_items_on_loan--;  // Decrement static member when an item is returned
    }

    void borrowItem() {
        copies_on_loan_++;
        total_items_on_loan++;  // Increment static member when an item is borrowed
    }

    int availableCopies() {
        return total_copies_ - copies_on_loan_;
    }

    static int getTotalItemsOnLoan() {  // Static function to query the total items on loan
        return total_items_on_loan;
    }

private:
    int total_copies_;
    int copies_on_loan_;
};

// Initialize the static member
int LibraryItem::total_items_on_loan = 0;
```

Now, DVD and Book sub-classes will share the static variable. As static variables are shared across all instances of a class.

(c)

i) Invariants for the LibraryItem Class

1. The number of **copies_on_loan_** should never exceed **total_copies_**.
2. The number of **copies_on_loan_** should never be negative.
3. The number of **total_copies_** should always be greater than or equal to zero.

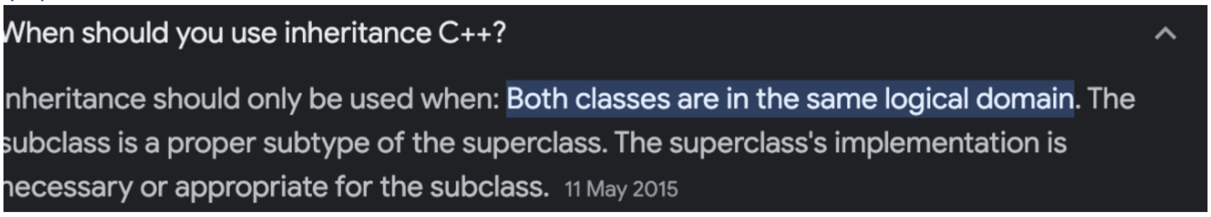
ii) Modifications to LibraryItem for Defensive Programming

To implement defensive coding practices that maintain these invariants, you can add checks in your methods to ensure that these conditions are always true. When a condition is violated, you can throw an exception or handle it in some other appropriate way.
Here's how you might modify the **LibraryItem** class:

```
#include <stdexcept> // for std::logic_error

class LibraryItem {
public:
   static int total_items_on_loan;

   LibraryItem(int total_copies) : total_copies_{total_copies}, copies_on_loan_{0} {
      if (total_copies < 0) {
         throw std::logic_error("Total copies cannot be negative");
      }
   }

   void returnItem() {
      if (copies_on_loan_ <= 0) {
         throw std::logic_error("No copies are on loan");
      }
      copies_on_loan_--;
      total_items_on_loan--;
   }

   void borrowItem() {
      if (copies_on_loan_ >= total_copies_) {
         throw std::logic_error("All copies are already on loan");
      }
      copies_on_loan_++;
      total_items_on_loan++;
   }

   int availableCopies() {
      int available = total_copies_ - copies_on_loan_;
      if (available < 0) {
         throw std::logic_error("Available copies cannot be negative");
      }
      return available;
   }
```

```
    static int getTotalItemsOnLoan() {
        return total_items_on_loan;
    }

private:
    int total_copies_;
    int copies_on_loan_;
};

int LibraryItem::total_items_on_loan = 0;
```

(d)


When should you use inheritance C++?

nheritance should only be used when: Both classes are in the same logical domain. The ubclass is a proper subtype of the superclass. The superclass's implementation is necessary or appropriate for the subclass. 11 May 2015

Book is a library item. (I.e., an item that can be borrowed, so is DVD).

In the given scenario, the use of inheritance seems to be a reasonable design decision. Here's why:

Advantages:

1. **Code Reusability**: **LibraryItem** serves as a base class that encapsulates common properties (**total_copies_** and **copies_on_loan_**) and behaviors (**borrowItem**, **returnItem**, **availableCopies**). This avoids duplication of the same logic in both **Book** and **DVD** classes.
2. **Single Responsibility**: **LibraryItem** is responsible for managing the "library item" aspect, such as loaning and returning, while **Book** and **DVD** can focus on what makes them unique (e.g., title, author for **Book**, and title, runtime for **DVD**).
3. **Polymorphism**: If you ever need to perform the same operation on different kinds of library items, you can handle them all as **LibraryItem** pointers or references, which simplifies the code.

Disadvantages:

1. **Rigidity**: Inheritance is a strong form of coupling. If the **LibraryItem** class needs to change, there's a good chance those changes will propagate to **Book** and **DVD**. This could make the system harder to maintain in the long run.
2. **Overhead**: Although minimal in this case, inheritance introduces a level of indirection, which could be a concern in performance-critical applications.
3. **Data Encapsulation**: If the base class (**LibraryItem**) has private data members that the derived classes (**Book**, **DVD**) need to manipulate directly, it can break encapsulation. However, this is not a concern in this specific scenario.

Given these points, the benefits of using inheritance appear to outweigh the disadvantages, at least with the given specifications. The code reusability, single responsibility principle, and potential for polymorphism make it a good fit for this situation.

(e)

For this question, we need to create local instances of the library item class to use composition and leverage the methods it already has.

```
class LibraryItem {
public:
   LibraryItem(int total_copies=1): total_copies_{total_copies}, copies_on_loan_{0} {}
   void returnItem() { copies_on_loan_--; }
   void borrowItem() { copies_on_loan_++; }
   int availableCopies() const { return total_copies_ - copies_on_loan_; }
private:
   int total_copies_;
   int copies_on_loan_;
};

class Book {
public:
   Book(const std::string& title, const std::string& author, int copies)
      : library_item_{copies}, title_{title}, author_{author} {}

   std::string title() const { return title_; }
   std::string author() const { return author_; }
   void returnItem() { library_item_.returnItem(); }
   void borrowItem() { library_item_.borrowItem(); }
   int availableCopies() const { return library_item_.availableCopies(); }

private:
   LibraryItem library_item_;
   std::string title_;
   std::string author_;
};

class DVD {
public:
   DVD(const std::string& title, int runtime, int copies)
      : library_item_{copies}, title_{title}, runtime_{runtime} {}

   std::string title() const { return title_; }
   int runtime() const { return runtime_; }
   void returnItem() { library_item_.returnItem(); }
   void borrowItem() { library_item_.borrowItem(); }
   int availableCopies() const { return library_item_.availableCopies(); }

private:
   LibraryItem library_item_;
   std::string title_;
   int runtime_; };
```

Question 3

(Not sure how to draw this)

1. Team Member 1 initializes a new repository. There are no commits yet.
2. Team Member 1 makes their first commit.
3. Team Member 1 makes a second commit.
4. Team Member 1 pushes their changes to GitHub.

At this point:

- Team Member 1's repo: Commit 2 (HEAD -> master), Commit 1
- GitHub repo: Commit 2 (HEAD -> origin/master), Commit 1
- Team Member 2's repo: Not yet initialized

5. Team Member 2 clones the repository from GitHub. This means they have the same commits as are on GitHub at this point.

Team Member 2's repo after clone: Commit 2 (HEAD -> master), Commit 1

6. Team Member 2 makes their first commit.

Team Member 2's repo after this commit: Commit 6 (HEAD -> master), Commit 2, Commit 1

7. Meanwhile, Team Member 1 makes another commit.
8. And yet another commit by Team Member 1.

Team Member 1's repo at this point: Commit 8 (HEAD -> master), Commit 7, Commit 2, Commit 1

9. Team Member 1 pushes their changes to GitHub.

GitHub repo after the push: Commit 8 (HEAD -> origin/master), Commit 7, Commit 2, Commit 1

10. Team Member 2 pulls the changes from GitHub. This will fetch the commits from GitHub and merge them into their local master branch. As there are no merge conflicts, this will result in a fast-forward merge.

Team Member 2's repo after the pull: Commit 8, Commit 7, Commit 6 (HEAD -> master), Commit 2, Commit 1

11. Team Member 2 then pushes their commit to GitHub.

GitHub repo after the push: Commit 6 (HEAD -> origin/master), Commit 8, Commit 7, Commit 2, Commit 1

So, to summarize:

- **Team Member 1's repo at the end**:
  - Commit 8 (HEAD -> master)
  - Commit 7
  - Commit 2
  - Commit 1
- **Team Member 2's repo at the end**:
  - Commit 6 (HEAD -> master)
  - Commit 8
  - Commit 7
  - Commit 2
  - Commit 1
- **GitHub repo at the end**:
  - Commit 6 (HEAD -> origin/master)
  - Commit 8
  - Commit 7
  - Commit 2

- Commit 1

This description can be visualized as a series of nodes (for commits) connected by lines (indicating the parent-child relationship between commits) for each repository.

Question 4

(a)

To test the **ShoppingCart** class, we need to consider all possible scenarios to ensure that the class behaves as expected.

```
TEST_CASE("ShoppingCart Tests") {
  Item tomatoSauce("All Gold Tomato Sauce", 20.0); // Example price
  Item bread("Bread", 15.0); // Example price

  // Test 1: Adding a single item and checking total
  SUBCASE("Test simple pricing with one item") {
    ShoppingCart cart;
    cart.addItem(tomatoSauce);
    CHECK(cart.total() == 20.0); // Expected total
  }

  // Test 2: Adding multiple items without any promotion
  SUBCASE("Test simple pricing with multiple items") {
    ShoppingCart cart;
    cart.addItem(tomatoSauce);
    cart.addItem(bread);
    CHECK(cart.total() == 35.0); // Expected total (20 + 15)
  }

  // Test 3: Three-for-Two promotion, without other items
  SUBCASE("Test three-for-two promotion alone") {
    ShoppingCart cart;
    cart.addItem(tomatoSauce);
    cart.addItem(tomatoSauce);
    cart.addItem(tomatoSauce); // Added 3 tomato sauce items

    createThreeForTwoPromotion(tomatoSauce, cart);
    CHECK(cart.total() == 40.0); // Expected total (only pay for 2 items)
  }

  // Test 4: Three-for-Two promotion mixed with other items
  SUBCASE("Test three-for-two promotion with other items") {
    ShoppingCart cart;
    cart.addItem(tomatoSauce);
    cart.addItem(tomatoSauce);
    cart.addItem(tomatoSauce); // Added 3 tomato sauce items
```

```
        cart.addItem(bread); // Added bread

        createThreeForTwoPromotion(tomatoSauce, cart);
        CHECK(cart.total() == 55.0); // Expected total (40 + 15)
    }

    // Test 5: More than three items with Three-for-Two promotion
    SUBCASE("Test more than three items with promotion") {
        ShoppingCart cart;
        for (int i = 0; i < 5; ++i) {
            cart.addItem(tomatoSauce); // Added tomato sauce 5 times
        }
        cart.addItem(bread); // Added bread

        createThreeForTwoPromotion(tomatoSauce, cart);
        CHECK(cart.total() == 95.0); // Expected total (20*4 + 15, because one tomato sauce is
free)
    }

    // Test 6: Less than three items with Three-for-Two promotion
    SUBCASE("Test less than three items with promotion") {
        ShoppingCart cart;
        cart.addItem(tomatoSauce);
        cart.addItem(tomatoSauce); // Added 2 tomato sauce items

        createThreeForTwoPromotion(tomatoSauce, cart);
        CHECK(cart.total() == 40.0); // Expected total (no discounts since we don't have three
items)
    }

    // Further tests can be added for other edge cases and scenarios
}
```

(b)

Complex approach:

1. Item Class

class Item {
private:
std::string _item_name;
double _price;
 public:
 Item(std::string name, double price) : _item_name(name), _price(price) {} std::string name() const { return _item_name; } double price() const { return _price; } };

2. Promotion Interface This will allow us to easily extend and add new promotion types.

class Promotion {
public: virtual double applyPromotion(const std::map<Item, int>& items) const = 0; };

3. ThreeForTwoPromotion Class

```
class ThreeForTwoPromotion :
 public Promotion {
private: Item _promotedItem;
public: ThreeForTwoPromotion(const Item& item) : _promotedItem(item) {} double
applyPromotion(const std::map<Item, int>& items) const override { if
(items.count(_promotedItem) > 0)
 { int count = items.at(_promotedItem); return (2 * (count / 3) + count % 3) *
_promotedItem.price(); } return 0; } };
```

4. ShoppingCart Class

```
class ShoppingCart {
private: std::map<Item, int> _items; // Keeps track of items and their counts
std::vector<std::shared_ptr<Promotion>> _promotions;
 public: ShoppingCart() {}
void addItem(const Item& item) { if (_items.count(item))
{ _items[item]++; }
else { _items[item] = 1; }
 }
void addPromotion(const std::shared_ptr<Promotion>& promotion) {
_promotions.push_back(promotion); }
 double total() const {
double totalCost = 0.0;
for (const auto& [item, count] : _items) { totalCost += item.price() * count; }
 for (const auto& promotion : _promotions) { totalCost -= promotion-
>applyPromotion(_items); // Subtracting the discounted amount } return totalCost; } };
```

Usage Example:

```
int main() {
Item tomatoSauce("All Gold Tomato Sauce", 20.0);
Item bread("Bread", 15.0); ShoppingCart cart; cart.addItem(tomatoSauce);
cart.addItem(tomatoSauce); cart.addItem(tomatoSauce); cart.addItem(bread);
std::shared_ptr<Promotion> promo
std::make_shared<ThreeForTwoPromotion>(tomatoSauce); cart.addPromotion(promo);
std::cout << "Total cost: " << cart.total() << std::endl; return 0; }
```

With this design:

- The **Promotion** interface allows for extensibility. New promotions can be added without changing existing classes.
- The **ShoppingCart** class focuses on its primary responsibilities and delegates the promotion application to the respective promotion objects.
- Information hiding and encapsulation principles are followed.

Adding future promotions, such as combo specials or percentage discounts, becomes straightforward. Simply create a new class implementing the **Promotion** interface and modify the shopping cart to include the new promotion.

(2)
```
#include <iostream>
#include <vector>
#include <unordered_map>

class Item {
private:
   std::string _name;
   double _price;

public:
   Item(std::string name, double price) : _name(name), _price(price) {}

   std::string name() const {
      return _name;
   }

   double price() const {
      return _price;
   }

   bool operator==(const Item& other) const {
      return _name == other._name && _price == other._price;
   }
};

namespace std {
   template <>
   struct hash<Item> {
      size_t operator()(const Item& item) const {
         return hash<std::string>()(item.name()) ^ hash<double>()(item.price());
      }
   };
}

class ShoppingCart {
private:
   std::unordered_map<Item, int> _items;
   std::unordered_map<Item, int> _threeForTwoItems;

public:
   void addItem(const Item& item) {
      if (_items.find(item) != _items.end()) {
         _items[item]++;
      } else {
         _items[item] = 1;
      }
```

```
    }

    void addThreeForTwoItem(const Item& item) {
        _threeForTwoItems[item] = 0;  // Initializing with zero for the promotion
    }

    double total() const {
        double totalCost = 0.0;

        for (const auto& [item, count] : _items) {
            if (_threeForTwoItems.find(item) != _threeForTwoItems.end()) {
                int discountCount = count / 3; // Calculate the discount for every third item
                totalCost += (count - discountCount) * item.price();
            } else {
                totalCost += count * item.price();
            }
        }

        return totalCost;
    }
};

int main() {
    Item tomatoSauce("All Gold Tomato Sauce", 20.0);
    Item bread("Bread", 15.0);

    ShoppingCart cart;
    cart.addItem(tomatoSauce);
    cart.addItem(tomatoSauce);
    cart.addItem(tomatoSauce);
    cart.addItem(bread);

    cart.addThreeForTwoItem(tomatoSauce); // Setting tomato sauce for the promotion

    std::cout << "Total cost: " << cart.total() << std::endl;

    return 0;
}
```
(3) The simplest implementation:
```
#include <iostream>
#include <vector>
#include <string>

class Item {
private:
    std::string _name;
    double _price;
```

```
public:
    Item(std::string name, double price) : _name(name), _price(price) {}

    std::string name() const {
        return _name;
    }

    double price() const {
        return _price;
    }
};

class ShoppingCart {
private:
    std::vector<Item> _items;
    std::vector<std::string> _threeForTwoItemNames;

public:
    void addItem(const Item& item) {
        _items.push_back(item);
    }

    void addThreeForTwoItem(const std::string& itemName) {
        _threeForTwoItemNames.push_back(itemName);
    }

    double total() const {
        double totalCost = 0.0;

        for (const auto& itemName : _threeForTwoItemNames) {
            int count = 0;
            for (const auto& item : _items) {
                if (item.name() == itemName) {
                    count++;
                }
            }
            int chargeableCount = count - (count / 3);  // Account for the discount
            for (const auto& item : _items) {
                if (item.name() == itemName) {
                    totalCost += chargeableCount * item.price();
                    break;
                }
            }
        }

        for (const auto& item : _items) {
            if (std::find(_threeForTwoItemNames.begin(), _threeForTwoItemNames.end(),
item.name()) == _threeForTwoItemNames.end()) {
                totalCost += item.price();
            }
        }

        return totalCost;
    }
};

int main() {
    Item tomatoSauce("All Gold Tomato Sauce", 20.0);
    Item bread("Bread", 15.0);

    ShoppingCart cart;
    cart.addItem(tomatoSauce);
    cart.addItem(tomatoSauce);
    cart.addItem(tomatoSauce);
    cart.addItem(bread);

    cart.addThreeForTwoItem("All Gold Tomato Sauce"); // Setting tomato sauce for the
promotion

    std::cout << "Total cost: " << cart.total() << std::endl;

    return 0;
}
```