



Final Review Worksheet Solutions

CPSC121 Fall 2022

JC Dy and Paul Salvador Inventado



Introduction

Wow, this semester has gone by really fast. But before you can finish up this class and beat the game, there is one last boss - the final! The final builds upon all the topics we've covered this semester, but emphasizes topics covered after the midterm exam.

This document is meant to provide you supplementary practice questions for topics covered after the midterm. Do not use this as a "be all end all" guide! It is still highly recommended that you review previous course material. You may also take the practice final exam on Canvas for additional practice with the test format.

With our time together coming to a close, I want you to know that I'm so incredibly grateful to have been your instructor this semester. Prof Paul and I are so proud to have seen how far you've grown since our first day together, and we see so much potential for many many more great things in the semesters and years to come.

As you continue on your journey here at Fullerton, remember that I'm rooting for you. If you ever forget to believe in yourself, just think back on all you've accomplished in just the few months we've spent together. If all else fails, [believe in me, who believes in you](#).

[Introduction](#)

[Composition](#)

[Object Composition](#), [Constructors](#), [Invariants](#)

[Dynamic Memory Allocation](#)

[Stack vs Heap](#), [Raw Pointers](#), [Shared Pointer](#)

[Iterators](#)

[Vector Iterators](#), [Map Iterators](#)

[Recursion](#)

[Recursive Objects](#), [Recursive Functions](#)

[RAII](#)

[Inheritance](#)

[Base Class](#), [Derived Class](#), [Function Overriding](#)

[Operator Overloading](#)

[Rule of Three](#)

[Exceptions](#)

[Templates](#)

[Function Templates](#), [Class Templates](#)

Composition

Object Composition

In object-oriented programming, **composition** describes how an object may contain other objects. Composition defines a “HAS-A” relationship. As an example, take this Post class, which is *composed of* other objects, to represent a social media post.

```
class Post {
public:
    Post(std::string name, std::string caption)
        : author_(name), caption_(caption) {}

    void AddComment(User user, std::string text) {
        if (!text.empty() && user.IsActive()) {
            Comment comment(text);
            comments_.push_back(comment);
        }
    }
    void EditCaption(std::string text) {
        caption_.SetText(text);
        caption_.SetWasEdited(true);
    }

private:
    User author_;
    Caption caption_;
    int like_count_;
    std::vector<Comment> comments_;
};
```

1. Which among the member variables are objects? Mark all that apply.

- ☐ **author_** ☐ **caption_** ☐ like_count_ ☐ **comments_**

2. Which of the following statements are true?

- ☐ **Post HAS-A Caption** ☐ User HAS-A Post ☐ Caption HAS-A User
☐ **Post HAS-A User** ☐ Post HAS-A Comment ☐ **Post HAS-A vector of Comments**

3. Which of the following functions must be defined in the object member variable classes for this code to compile? Select all that apply.

- ☐ Caption::EditCaption ☐ **Caption::SetText** ☐ **Caption::SetWasEdited**
☐ User::SetName ☐ **User::IsActive** ☐ Comment::SetText ☐ User::Post

Constructors

Consider the non default constructor for the Post class:

```
Post(std::string name, std::string caption)
    : author_(name), caption_(caption) {}
```

4. In the member initializer list we pass an argument (`std::string name`) to construct the `author_` object member variable. What happens to the string?
 - ☐ `author_` is initialized to the value stored in `name`.
 - ☐ `author_` is initialized to a new `User` object set to the value stored in `name`.
 - ☒ `author_` is initialized to a new `User` object constructed with the non-default `User` constructor that accepts a string.
 - ☐ `author_` is initialized to a new `User` object constructed with the default `User` constructor.
5. Which `Caption` constructor is called by the `caption_` initializer?
 - ☐ `Caption();`
 - ☒ `Caption(std::string caption);`
 - ☐ `Caption(const Caption &other);`
6. Notice that we don't explicitly initialize the `comments_` vector in the member initializer list. Which of the following is true? Mark all that apply.
 - ☐ `comments_` is initialized to the default value `nullptr`.
 - ☒ `comments_` is initialized to a new `std::vector` object with no elements.
 - ☒ `comments_` is initialized by an implicit call to the default constructor.
 - ☐ `comments_` is not initialized when a `Post` object is instantiated.

Invariants

A class invariant specifies what conditions should be true over the course of an object's lifetime. Consider the following class and what invariants should be held for its objects:

```
class Movie {
public:
    void SetTitle(std::string title);
    void SetReleaseDate(int month, int day, int year);
    void SetBudget(double budget);

private:
    std::string title_;
    Date release_date_;
    double budget_;
};
```

1. What class invariants could be held for `Movie` instances? Mark all that apply.
 - ☒ **title_ must be initialized to a non-empty string.**
 - ☒ **release_date_ must have a valid month, day, and year set.**
 - ☒ **budget_ must be set to a non-negative value.**
2. Assume we want to enforce the invariant on `budget_` as stated above. Which of the following conditions could be added to `SetBudget` to enforce the invariant?
 - ☐ `if (budget > 0) { budget_ = budget; }`
 - ☐ `if (budget_ >= 0) { budget = budget_; }`
 - ☒ **`if (budget >= 0) { budget_ = budget; }`**
3. Complete the `SetTitle` function such that the `title_` is only set if the title is non-empty. Print an error message (or throw an exception) if it is.

```
void SetTitle(std::string title) {

    if (title.empty()) {
        throw std::invalid_argument("Title must not be empty");
    } else {
        title_ = title;
    }

}
```

4. Complete the SetReleaseDate function such that the release_date_ is only set if the month, day, and year are valid.

```
void SetReleaseDate(int month, int day, int year) {  
  
    if (month < 1 || month > 12) {  
        throw std::invalid_argument("Invalid month");  
    } else if (day < 1 || day > 31) {  
        throw std::invalid_argument("Invalid day");  
    } else if (year < 0) {  
        throw std::invalid_argument("Invalid year");  
    }  
  
    release_date_ = Date(month, day, year);  
}
```

You decide to add this accessor function for the ReleaseDate member variable:

```
Date& GetReleaseDate();
```

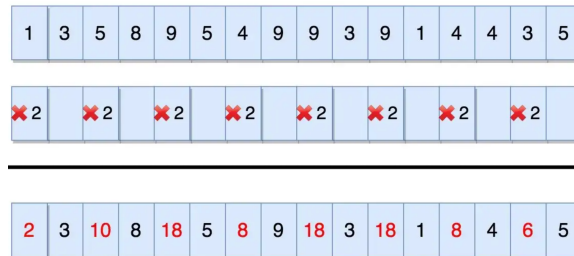
5. What is the effect of the & symbol on the return type?
- ☐ GetReleaseDate returns a copy of the Date object (pass-by-value), so the original release_date_ object cannot be accessed.
 - ☒ GetReleaseDate returns a reference of the Date object (pass-by-reference), so the original release_date_ object can be accessed.
6. To your dismay, you find that the release_date_ invariant is somehow broken after running your program. What is the flaw with this accessor function?
- ☐ Returning a copy of the Date object, is memory expensive and time inefficient.
 - ☒ Returning a reference to the original Date object allows the original object to be mutated.
 - ☐ The return type uses the & symbol.
7. Write a new function declaration (i.e. you don't need to write the function body) for GetReleaseDate so that it allows users to view the object, but not mutate it.

```
const Date& GetReleaseDate() const;
```

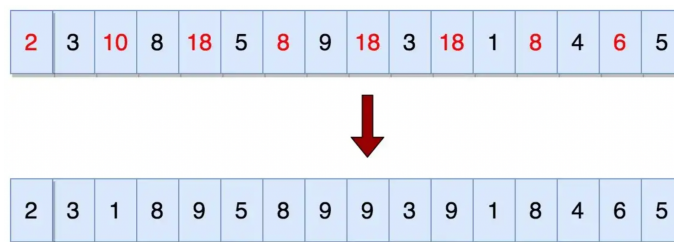
Challenge: Credit Card Number Validator

The Luhn Algorithm (https://en.wikipedia.org/wiki/Luhn_algorithm) is used to validate that a given credit card number is considered valid. It is usually used as a simple method of distinguishing valid credit card numbers from mistyped or otherwise incorrect numbers. The Luhn algorithm is as follows:

1. Starting from the rightmost digit, double every second digit. See the diagrams below for an example credit card number to validate (1358 9549 9391 4435)




2. If the result is greater than 10, i.e. if the result is a two-digit number add the individual digits to make it a single digit.



3. Add all the digits together and find the sum. If the sum is a multiple of 10, the credit card number is valid.




$$2 + 3 + 1 + 8 + 9 + 5 + 8 + 9 + 9 + 3 + 9 + 1 + 8 + 4 + 6 + 5 = 90$$

Because the digits sum up to 90, which is a multiple of 10, the credit card number “1358 9549 9391 4435” is valid.

On the next page, implement `IsCreditCardValid`, which accepts a credit card number and returns true if, according to the Luhn algorithm, this number is valid.

```
bool IsCreditCardValid(long n) {  
  
    int s{},c,t{};  
    for(;n>0;n/=10){c=n%10;if(t)c*=2;c=c%10+c/10;s+=c;t=!t;}  
    return !(s%10);  
  

```

ALTERNATIVE SOLUTION:

```
int c=0,u=0,m;  
for(;n;n/=10)m=n%10<<u++%2,c+=m%10+m/10;  
return!(c%10);  
  

```

```
}
```


Dynamic Memory Allocation

Stack vs Heap

Assuming the following lines of code are run, where in memory is the value associated with each expression found?

```
int spice = 2;
int* spice_ptr = &spice;
Fish fry(5);
Fish* chip = new Fish(5);
std::shared_ptr<Shark> shark_ptr = std::make_shared<Shark>();
```

1. spice
☒ **Stack** ☐ Heap ☐ Invalid
2. *spice
☐ Stack ☐ Heap ☒ **Invalid**
3. spice_ptr
☒ **Stack** ☐ Heap ☐ Invalid
4. *spice_ptr
☒ **Stack** ☐ Heap ☐ Invalid
5. fry
☒ **Stack** ☐ Heap ☐ Invalid
6. *fry
☐ Stack ☐ Heap ☒ **Invalid**
7. chip
☒ **Stack** ☐ Heap ☐ Invalid
8. *chip
☐ Stack ☒ **Heap** ☐ Invalid
9. shark_ptr
☒ **Stack** ☐ Heap ☐ Invalid
10. *shark_ptr
☐ Stack ☒ **Heap** ☐ Invalid

Raw Pointers

1. The `new` keyword returns a pointer to memory that has been allocated. Fill in the blank below to complete the function definition.

```
___ Fish* _____ CreateFish(std::string name) {  
    return new Fish(name);  
}
```

2. The `new` keyword is used to dynamically allocate memory in the heap. What **keyword** is used to deallocate (free up) memory that was used?
☐ `destructor` ☐ `del`
☒ **`delete`** ☐ `remove`
3. A memory leak is a common problem when handling raw pointers. Which of the following could cause a memory leak?
☐ Dereferencing a pointer that was never initialized or points to invalid memory
☒ **Forgetting to delete memory that was allocated on the heap.**
☐ Accessing an index in a vector that is out of bounds.

Shared Pointer

We prefer to use smart pointers like `shared_ptr` over raw pointers because `shared_ptr` prevents memory leaks by automatically deleting unused objects from memory.

1. Which of the following would properly initialize `game_ptr`?
`std::shared_ptr<Game> game_ptr = _____`
☐ `new Game();` ☒ **`std::make_shared<Game>();`**
☐ `&Game();` ☐ `std::shared_ptr<Game>();`
2. Assume the `Game` class has a member function called `Play`. Which of the following statements would correctly invoke `Play`? Mark all that apply.
☒ **`(*game_ptr).Play();`** ☐ `*game_ptr.Play();`
☐ `(*game_ptr)->Play();` ☒ **`game_ptr->Play();`**
3. Create a `shared_ptr` of a `Game` object named "tuffle", using the non-default constructor for `Game` below. You may use any values to initialize.

```
Game(std::string title, double win_rate, bool is_favorite);
```

```
std::shared_ptr<Game> tuffle =  
    std::make_shared<Game>("Tuffle", 100.0, true);
```

Iterators

Vector Iterators

Below each line, write the value of the expression on the left after executing the statement. The first two have been done for you.

1. `std::vector<string> cities {"San Diego", "Whittier", "Irvine"};`

<code>cities:</code>	<code>{"San Diego", "Whittier", "Irvine"}</code>
----------------------	--

2. `cities.push_back("Tustin");`
`cities.push_back("Chino");`

<code>cities.size():</code>	<code>5</code>
-----------------------------	----------------

3. `std::vector<string>::iterator it = cities.begin();`

<code>*it:</code>	<code>"San Diego"</code>
-------------------	--------------------------

4. `it++;`

<code>*it;</code>	<code>"Whittier"</code>
-------------------	-------------------------

5. `string city = *(cities.begin() + 4);`

<code>city</code>	<code>Chino</code>
-------------------	--------------------

6. `city = "Barcelona";`

<code>cities</code>	<code>{"San Diego", "Whittier", "Irvine", "Tustin", "Chino"}</code>
---------------------	---

7. `for (int i = 0; i < 3; i++) {`
 `cities.erase(cities.begin());`
}

<code>cities.size():</code>	<code>2</code>
-----------------------------	----------------

8. `std::vector<string>::iterator it2 = cities.begin();`

<code>*it2:</code>	<code>"Tustin"</code>
--------------------	-----------------------

9. `it2++;`
`it2++;`
`bool reached_end = (it2 == cities.end());`

<code>reached_end:</code>	<code>true</code>
---------------------------	-------------------

Map Iterators

Below each line, write the value of the expression on the left after executing the statement. The first two have been done for you.

1. `std::map<string, int> pet_ages;`

pet_ages:	{ }
-----------	-----

2. `pet_ages.insert({"Jasmine", 8});`
`pet_ages.insert({"Bitzy", 9});`
`pet_ages.insert({"Toby", 1});`

pet_ages:	{{"Jasmine", 8}, {"Bitzy", 9}, {"Toby", 1}}
-----------	---

3. `std::map<string, int>::iterator it = pet_ages.begin();`

it->first:	Jasmine
it->second:	8

4. `it++;`

it->first:	Bitzy
it->second:	9

5. `it++;`
`int age = it->second;`

age:	1
------	---

6. `age++;`

it->second:	1
-------------	---

7. `std::map<string, int>::iterator it2 = pet_ages.find("Toby");`
`if (it2 != pet_ages.end()) {`
 `pet_ages.erase(it2);`
`}`

pet_ages.size():	2
------------------	---

8. `int count = pet_ages.count("Jasmine");`

count:	1
--------	---

9. `count = pet_ages.count("Toby");`

count:	0
--------	---

Challenge: RemoveRepeated

Implement the **RemoveRepeated** function, which takes in a vector **nums** and an integer **k** and returns the first value that appears **k** times in a row. Erase one of those values from the vector.

Note: you can assume that the vector will have a value that appears at least **k** times in a row.

For example, assume we have a vector {10, 9, 10, 9, 9, 10, 8, 8, 8, 7}.

- With **k** set to 2, the return value of **RemoveRepeated** would be 9, and the resulting vector would be {10, 9, 10, 9, 10, 8, 8, 8, 7}.
- With **k** set to 3, the return value of **RemoveRepeated** would be 8, and the resulting vector would be {10, 9, 10, 9, 9, 10, 8, 8, 7}.

```
int RemoveRepeated(std::vector<int> &nums, int k) {  
  
    int count = 1, last = 0;  
  
    for (std::vector<int>::iterator it = nums.begin(); it != nums.end(); it++) {  
        if (*it == last) {  
            count++;  
        } else {  
            last = *it;  
            count = 1;  
        }  
  
        if (count == k) {  
            nums.erase(it);  
            return last;  
        }  
    }  
    return -1;  
}
```

```
}
```

Recursion

Recursive Objects

A recursive object is an object that has member(s) of the same object type.
code. For example, consider the Book class, which represents a book with an author and pages within it:

```
class Page {
public:
    Page(std::vector<std::string> contents,
         std::shared_ptr<Page> next_page);
    std::shared_ptr<Page> GetNextPage() { return next_page_; }
    std::vector<std::string> GetContents() { return contents_; }

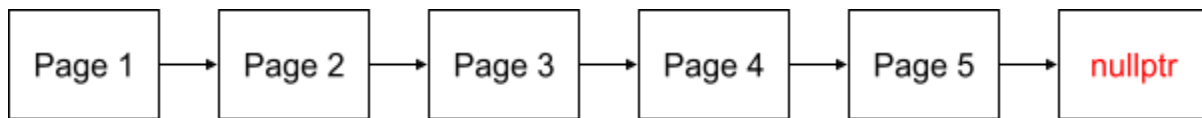
private:
    // Represents the next page.
    std::shared_ptr<Page> next_page_;
    // A list of the words on the page.
    std::vector<std::string> contents_;
};

class Book {
public:
    Book(std::string author);
    std::shared_ptr<Page> GetFirstPage() { return first_page_; }

private:
    // Represents the author of the book.
    std::string author_;
```

```
// Represents the first page in the book.  
std::shared_ptr<Page> first_page_  
};
```

1. Which of the following is the recursive object?
☐ Book ☐ Page ☐ std::vector<std::string>
2. The recursive structure forms a sequential data structure called a LinkedList, as seen below. In the final box, write the keyword that indicates that no next_page_ exists.



Recursive Functions

1. Implement a recursive function that returns the total number of pages.

```
int Page::TotalPages() {  
    if (next_page_ == nullptr) {  
        return 1;  
    }  
  
    return 1 + next_page_->TotalPages();  
  
}
```

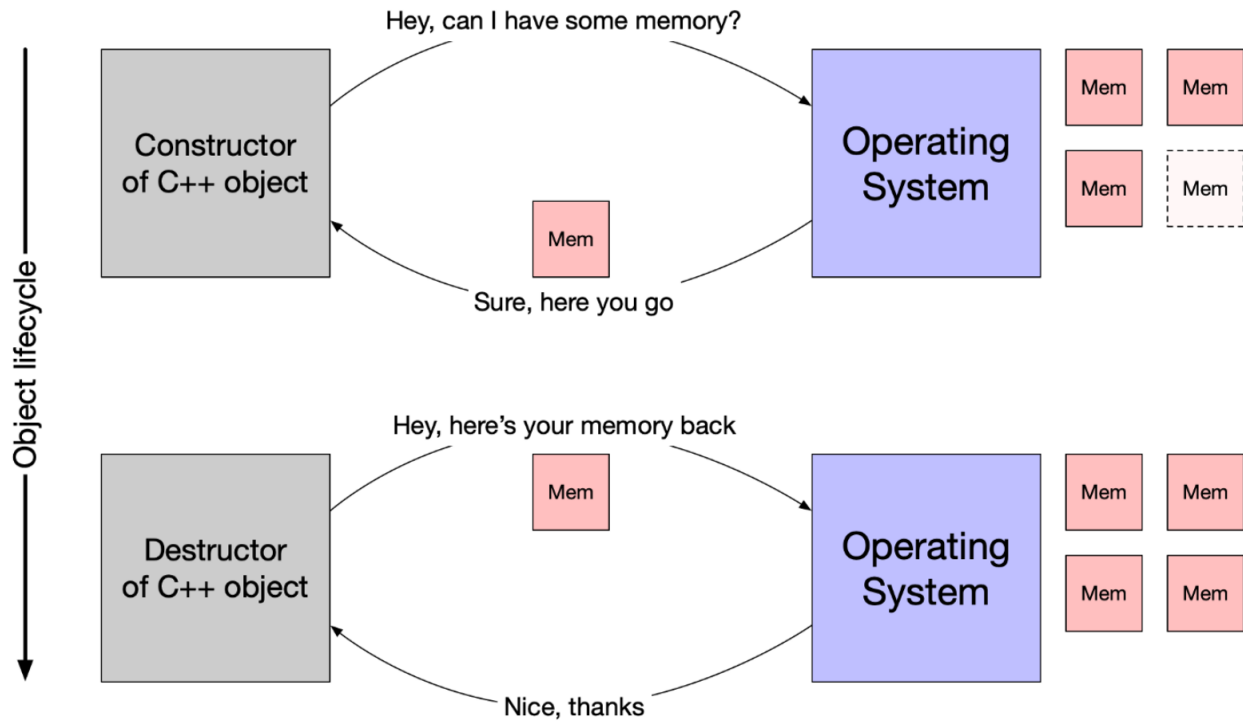
2. Implement a function that returns the frequency of a given word, starting at the current page and including all the following pages.

```
int Page::WordFrequency(const std::string &word) {  
    int total = 0;  
  
    for (const std::string &content : contents_) {  
        if (content == word) {  
            total++;  
        }  
    }  
  
    if (next_page_ == nullptr) {  
        return total;  
    }  
  
    return total + next_page_->WordFrequency(word);  
}
```

RAII

RAII is built upon two principles:

1. Allocating memory happens when an object is initialized (in the constructor)
2. Freeing memory happens when an object is destroyed (in the destructor)



1. RAII (Resource Acquisition is Initialization) centers around the idea that acquiring resources should occur in the _____, and that those resources should be released in the _____.

☒ constructor, destructor ☐ destructor, constructor ☐ new, delete ☐ delete, new

2. A destructor works opposite to the constructor - it destructs the objects. Which of the following is the destructor for the VideoGame class?

☐ VideoGame(); ☐ delete(); ☐ Destruct(); ☒ ~VideoGame();

3. When is a destructor called? Mark all that apply.

- ☒ A stack object's destructor is called when it goes out of scope (e.g. when its containing function exits, or the containing loop completes)
- ☐ An object's destructor is called when the heap runs out of memory.
- ☒ A heap object's destructor is called when an explicit call to delete is made.

Inheritance

Base Class, Derived Class

Inheritance represents an "IS-A" relationship. In this example, a CoffeeShop "IS-A" Shop, and thus inherits Shop's members, but also defines its own specialized members.

```
class Shop {
public:
    Shop(std::string name) : name_(name) {}
    std::string GetName() const { return name_; }
    void Print() const {
        std::cout << "Welcome to " << name_ << "!";
    }

private:
    std::string name_;
};

class CoffeeShop : public Shop {
public:
    CoffeeShop(std::string name, Menu menu, bool indoor,
               std::vector<Barista> staff)
        : Shop(name), menu_(menu),
          has_indoor_dining_(indoor), employees_(staff) {}

private:
    Menu menu_;
    bool has_indoor_dining_;
    std::vector<Barista> employees_;
};
```

1. Identify the base class and derived class in the above example.
 - ☒ Shop is the base class, and CoffeeShop is the derived class
 - ☐ CoffeeShop is the base class, and Shop is the derived class
2. Derived classes inherit all member variables and member functions from a base class without rewriting them. Which members are accessible from the CoffeeShop class? Mark all that apply.
 - ☒ GetName
 - ☒ Print
 - ☐ name_

3. Observe that the `CoffeeShop` constructor invokes the base class constructor via a call to `Shop(name)` in the member initializer list. Which of the following statements are true? Mark all that apply.
- ☐ The `CoffeeShop` class can invoke the `Shop` constructor in the member initializer list of its own constructor to initialize the members of the base class.
 - ☐ Omitting the call to the `Shop` constructor in the member initializer list of the `CoffeeShop` constructor would be acceptable as the `Shop` default constructor would be implicitly invoked.
 - ☐ The `Shop` class must be constructed first, so that `CoffeeShop` can inherit the member variables and functions from its base class.
4. The `CoffeeShop` class inherits only the public members of the `Shop` class.
- ☐ true ☐ false

Function Overriding

Function overriding is when a derived class redefines a function that was defined in the base class. Function overriding allows derived classes to create more specialized versions of a function, while being able to reuse what was defined in the base class.

5. Fill in the blank below to reuse the `Print` function that has already been defined in the `Shop` class.

```
void CoffeeShop::Print() const {  
    // Invoke the base class Print function  
  
    Shop::Print();  
  
    if (has_indoor_dining_) {  
        std::cout << "We're open for indoor dining!";  
    }  
}
```

6. Why is it necessary to specify the `<Base class>::` prefix to invoke an overridden member function in the base class?
- ☐ The `<Base class>::<member function name>` syntax resolves ambiguity between which function to call (`CoffeeShop::Print` vs `Shop::Print`)
 - ☐ The `<Base class>::<member function name>` syntax allows access to member functions that otherwise would be private to the derived class.

Operator Overloading

C++ allows us to make operators (e.g. +, -, *, ==, etc) work for objects of our own classes. The code below overloads the + operator for the Bottle class, allowing the liquid contents of two Bottle objects to be “added” together:

```
class Bottle {
public:
    Bottle(int volume) : volume_(volume) {}
    int GetVolume() const { return volume_; }

    Bottle operator+(const Bottle &other) {
        Bottle combined(volume_ + other.GetVolume());
        return combined;
    }

private:
    int volume_;
};
```

1. Write down the output to the console after running the following lines of code:

```
Bottle sprite(12);
Bottle lemonade(6);
Bottle c = sprite + lemonade;
```

std::cout << c.GetVolume();	18
std::cout << sprite.GetVolume();	12

2. Write down the output to the console after running the following lines of code:

```
Bottle half(5);
Bottle whole = half + half;
```

std::cout << whole.GetVolume();	10
std::cout << half.GetVolume();	5

3. Which of the following statements is equivalent to:

```
Bottle calimochu = wine + coke;
```

- ☐ Bottle calimochu.operator+(wine, coke);
- ☐ Bottle calimochu = operator+(wine, coke);
- ☒ Bottle calimochu = wine.operator+(coke);
- ☐ Bottle calimochu = wine.+(coke);

The copy assignment operator overload is shown below. It is called when we assign a `Bottle` object to a `Bottle` variable.

4. Fill in the blank to assign the volume appropriately when invoking the copy assignment operator.

```
Bottle operator=(const Bottle &other) {  
    volume_ = other.GetVolume();  
    return *this;  
}
```

5. Write the operator overload for the `<<` symbol such that the following statements transfer over all the contents of `cup` to `jug` (i.e. after the statements run, `cup` will have a volume of 0, and `jug` would have a volume of 20).

```
Bottle cup(2);  
Bottle jug(18);  
jug << cup;
```

```
void operator<<(const Bottle &other) {  
    volume_ += other.GetVolume();  
    other.volume_ = 0;  
  
}
```

Rule of Three

The Rule of Three states that if you define one of the following, then your class almost certainly requires all three. Select exactly 3.

- | | |
|--|---|
| <input type="checkbox"/> Destructor | <input type="checkbox"/> Default constructor |
| <input type="checkbox"/> Member initializer list constructor | <input type="checkbox"/> Copy constructor |
| <input type="checkbox"/> Copy assignment operator | <input type="checkbox"/> Non-default destructor |
| <input type="checkbox"/> Equality operator | <input type="checkbox"/> Constructor overload |

Exceptions

1. Which keyword is used to indicate a block of code which may throw an exception?
☒ try ☐ catch ☐ throw ☐ except
2. Which keyword do we use to tell the compiler that an exception occurred?
☐ try ☐ catch ☒ throw ☐ except
3. Which keyword do we use to indicate the code that should run in the event of an exception?
☐ try ☒ catch ☐ throw ☐ except

```
bool Bank::IsActiveAccount(int account_id) {...}

void Bank::Deposit(double amount, int account_id) {
    if (_____4_____) {
        throw std::invalid_argument("Deposit must be at least $0.00");
    } else if (_____5_____) {
        throw std::invalid_argument("Account is not active.");
    }
}
```

4. Which condition would best fit the error description that would be thrown in the first if condition?
☐ amount > 0 ☒ amount < 0 ☐ amount <= 0 ☐ amount >= 0
5. Which condition would best fit the error description that would be thrown in the else if condition?
☐ IsActiveAccount(account_id) ☒ !IsActiveAccount(account_id)
☐ !(account_id >= 1) ☐ account_id.IsActive == false
6. Complete the try catch block to handle the exception Deposit may throw.

```
try {
    Bank b;
    double amount;
    std::cin >> amount;
    b.Deposit(amount, 12345678);
    std::cout << "Cha-ching!";
} catch (const std::invalid_argument &e) {
    std::cout << e.what() << "\n";
}
```

7. Give a sample input such that "Cha-ching!" is **NOT** printed out.
amount: -1 (any negative number works here)

Templates

Function Templates

A C++ template defines a blueprint for creating new types of functions. Templates are a way to write more generic code that works for many different types.

1. Which line defines a template parameter T that may be used in a template function?

☐ typename <template T> ☐ template <typedef T>
☒ template <typename T> ☐ template <T>

2. Template parameters are always named using the keyword “T”.

☐ true ☒ false

Consider the function template below:

```
template <typename T>
T GetMax(T x, T y) {
    T result;
    // ? and : together form the “ternary operator”, which is
    // equivalent to: if (x > y) return x, else y
    // (condition) ? (true_value) : (false_value)
    result = x > y ? x : y;
    return result;
}
```

3. When the GetMax function is invoked, the template parameter T will be replaced with the concrete data type of the argument passed in. What must be true about any data type substituted for T in the GetMax function? Mark all that apply.
- ☐ The operator? overload must be defined for the T data type.
 - ☒ The operator> overload must be defined for the T data type.
 - ☒ A default constructor must be defined for a non-primitive T data type.

Class Templates

C++ templates are not just limited to functions - they can be applied to classes too.

Consider the Box class template below:

```
template <typename T>
class Box {
public:
    void SetValue(T value) { value_ = value; }

private:
    T value_;
};

int main() {
    Box<int> red_gift;
    Box<double> blue_gift;
    Box<std::string> green_gift;

    red_gift.SetValue(____a____);
    blue_gift.SetValue(____b____);
    green_gift.SetValue(____c____);
}
```

1. The **Box** class is a template class whose template parameter is T, and T can be substituted anywhere throughout the class where a data type normally would be.

☒ true

☐ false

2. What set of values could be substituted in the blanks (a, b, c, see above) to complete the main function such that the program compiles? Mark all that apply.

☐ a: 12, b: 25, c: "happy holidays!"

☐ a: 1, b: 1.0, c: "happy new year!"

☐ a: 100, b: 121, c: "good luck on finals!"

3. Because **Box** is a template class, we can put any type of data inside without having to rewrite the class every time we want to support a new data type.

Let us know what's on your wish list this holiday season and instantiate a Box of any data type, called `secret_santa_gift` below. Then, call `SetValue` to put your 🎁 gift 🎁 inside!

```
Box<Chocolate> secret_santa_gift;
```

```
secret_santa_gift.SetValue(Chocolate("ferrero rocher"));
```