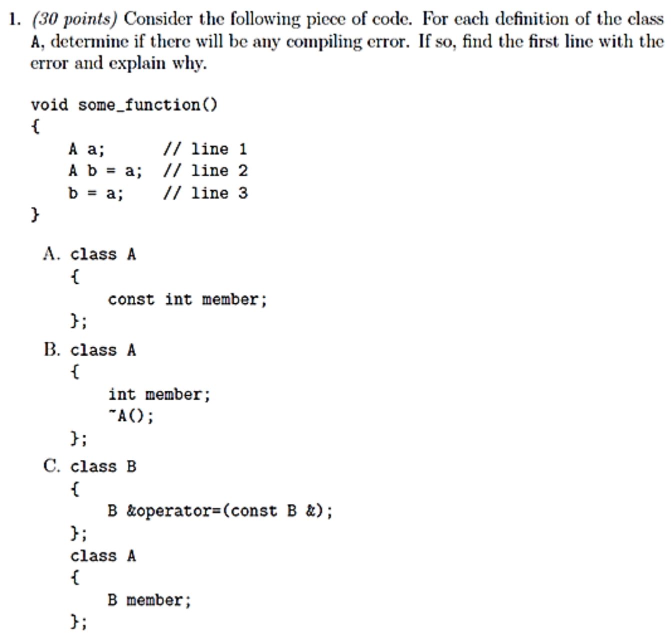
HOMEWORK-4

OBJECT ORIENTED PROGRAMMING AND MACHINE LEARNING

NAME : Meenakshi Sridharan Sundaram
HAWK ID: A20492581
SUBJECT CODE: ECE-449/590
DUE DATE : Wednesday, 27th November 2024
PROFESSOR NAME: Dr. Won Jae Yi

QUESTION 1



1. *(30 points)* Consider the following piece of code. For each definition of the class A, determine if there will be any compiling error. If so, find the first line with the error and explain why.

```
void some_function()
{
    A a;      // line 1
    A b = a;  // line 2
    b = a;    // line 3
}
```

A.
```
class A
{
    const int member;
};
```

B.
```
class A
{
    int member;
    ~A();
};
```

C.
```
class B
{
    B &operator=(const B &);
};
class A
{
    B member;
};
```

a) In question 1 as mentioned above, we have to analyse the some_function() under three different class definitions to find if there are any compilation errors and explain the reasons behind them

- Issue: For class A, the issue is in the member const int member, where it is a const variable that must be initialised at the time of the object construction. However, no constructor is defined in the class; the compiler generates a default constructor that makes the member uninitialized. This ends up causing a compilation error.
- Error: In line 1, A a, results in a compilation error because the const member is not initialized
- Explanation: A const variable cannot be left uninitialized and since there is no user-defined constructor present to initialize it, it causes an occurrence of compilation error

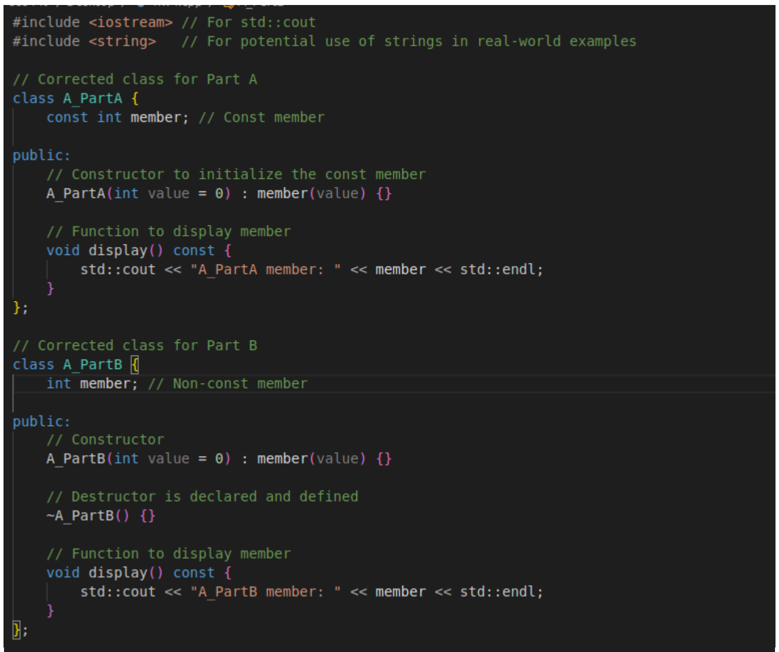
b) The issue in class A

- The destructor in class A - ~A() is declared but no definition for it is present. When the object a goes out of scope at the end of some_function(), the compiler tries to invoke the destructor but since it is undefined it cannot find its definition. This causes a linker error
- Error: There is no presence of compilation error in lines 1,2 or 3 during the building. But a linker error is caused when the destructor is called
- Explanation: Since the destructor is declared but not defined, it leads to a missing symbol error at the linking stage

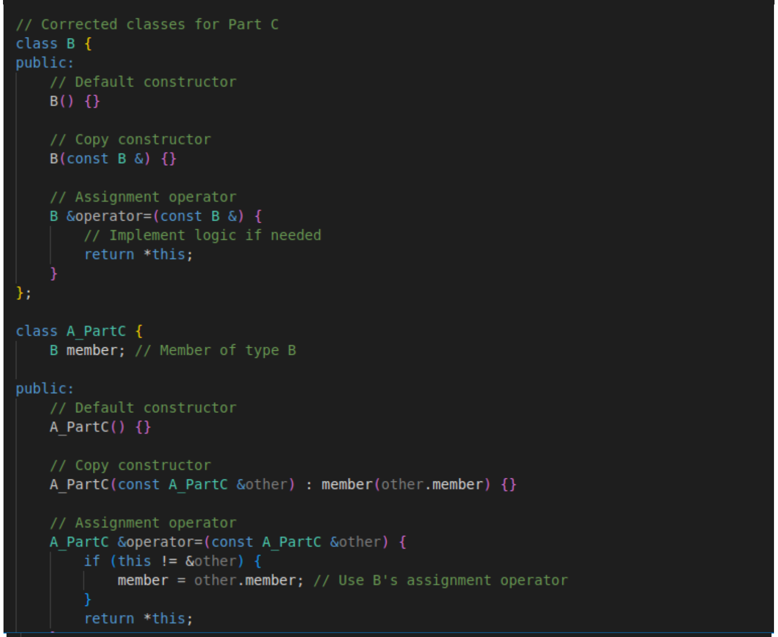
c) In class B, an assignment operator is declared but is not defined. Since A contains a member of type B, any assignment to an object of A(such as b=a on line 3) requires the assignment operator for B. This again results in a linker error because B's assignment operator is declared but not defined.

- Error: In line 3, b=a; results in a linker error because B's assignment operator is missing
- Explanation: The assignment operator for class A relies on the assignment operator for class B, which is declared but not defined anywhere.

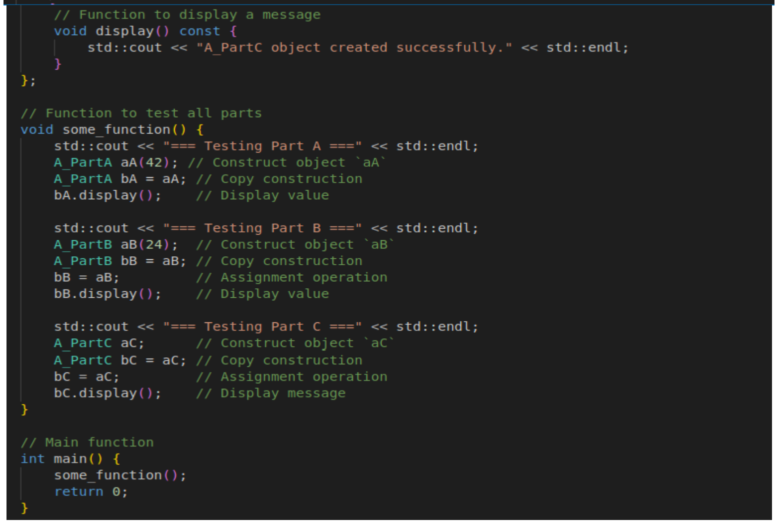
Here is the fully corrected code and its expected output.



```cpp
#include <iostream> // For std::cout
#include <string>   // For potential use of strings in real-world examples

// Corrected class for Part A
class A_PartA {
    const int member; // Const member

public:
    // Constructor to initialize the const member
    A_PartA(int value = 0) : member(value) {}

    // Function to display member
    void display() const {
        std::cout << "A_PartA member: " << member << std::endl;
    }
};

// Corrected class for Part B
class A_PartB {
    int member; // Non-const member

public:
    // Constructor
    A_PartB(int value = 0) : member(value) {}

    // Destructor is declared and defined
    ~A_PartB() {}

    // Function to display member
    void display() const {
        std::cout << "A_PartB member: " << member << std::endl;
    }
};
```

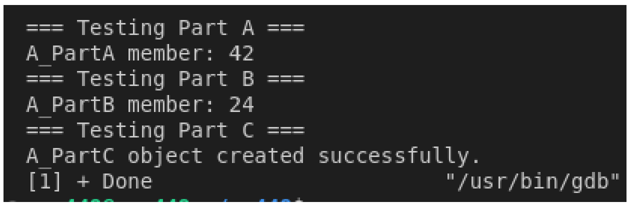


```cpp
// Corrected classes for Part C
class B {
public:
    // Default constructor
    B() {}

    // Copy constructor
    B(const B &) {}

    // Assignment operator
    B &operator=(const B &) {
        // Implement logic if needed
        return *this;
    }
};

class A_PartC {
    B member; // Member of type B

public:
    // Default constructor
    A_PartC() {}

    // Copy constructor
    A_PartC(const A_PartC &other) : member(other.member) {}

    // Assignment operator
    A_PartC &operator=(const A_PartC &other) {
        if (this != &other) {
            member = other.member; // Use B's assignment operator
        }
        return *this;
```



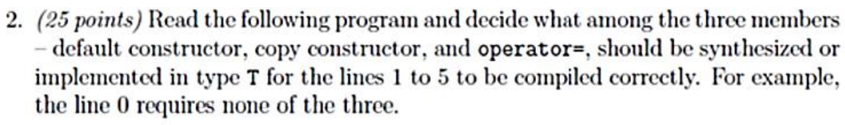
```cpp
    // Function to display a message
    void display() const {
        std::cout << "A_PartC object created successfully." << std::endl;
    }
};

// Function to test all parts
void some_function() {
    std::cout << "=== Testing Part A ===" << std::endl;
    A_PartA aA(42); // Construct object `aA`
    A_PartA bA = aA; // Copy construction
    bA.display();    // Display value

    std::cout << "=== Testing Part B ===" << std::endl;
    A_PartB aB(24);  // Construct object `aB`
    A_PartB bB = aB; // Copy construction
    bB = aB;         // Assignment operation
    bB.display();    // Display value

    std::cout << "=== Testing Part C ===" << std::endl;
    A_PartC aC;      // Construct object `aC`
    A_PartC bC = aC; // Copy construction
    bC = aC;         // Assignment operation
    bC.display();    // Display message
}

// Main function
int main() {
    some_function();
    return 0;
}
```

OUTPUT OF QUESTION 1

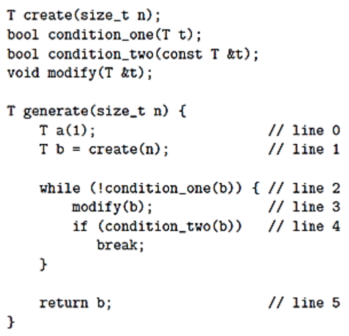


```
=== Testing Part A ===
A_PartA member: 42
=== Testing Part B ===
A_PartB member: 24
=== Testing Part C ===
A_PartC object created successfully.
[1] + Done                          "/usr/bin/gdb"
```

QUESTION 2:



2. *(25 points)* Read the following program and decide what among the three members – default constructor, copy constructor, and **operator=**, should be synthesized or implemented in type **T** for the lines 1 to 5 to be compiled correctly. For example, the line 0 requires none of the three.



```
T create(size_t n);
bool condition_one(T t);
bool condition_two(const T &t);
void modify(T &t);

T generate(size_t n) {
    T a(1);                      // line 0
    T b = create(n);             // line 1

    while (!condition_one(b)) {  // line 2
        modify(b);               // line 3
        if (condition_two(b))    // line 4
           break;
    }

    return b;                    // line 5
}
```

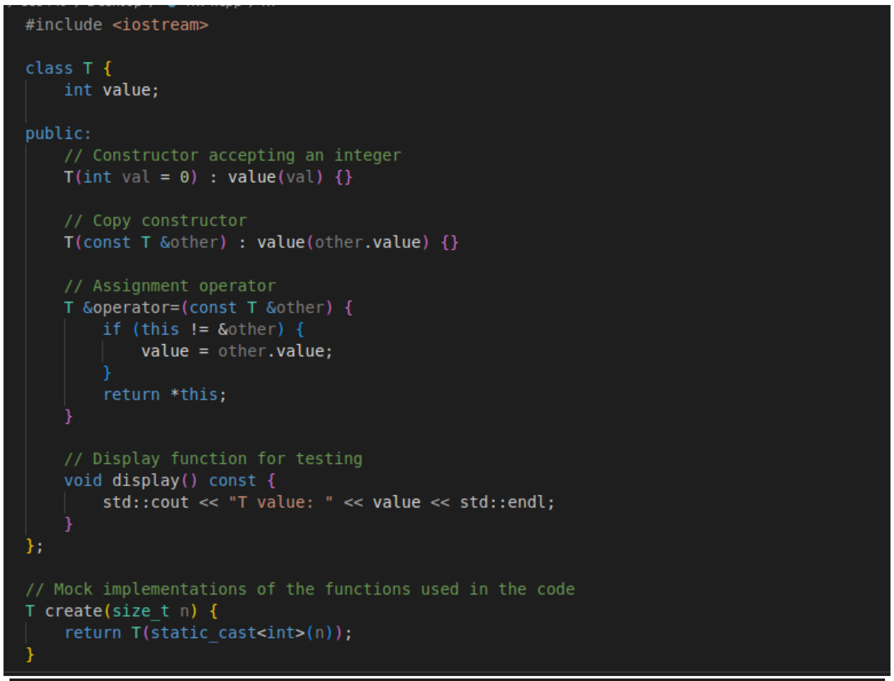
For question 2 we need to analyze the code and check with which member functions needs to be implemented explicitly to avoid any compilation present. The program so far includes a default constructor, copy constructor and assignment operator.

- **a)** In line 0 which has T a(1) requires a constructor that accepts one argument hence no special members are needed here
- **b)** In line 1 which is T b=create(n), create(n) returns an object which is of type T. Hence the initialization of T b=create(n) requires a copy contructor to copy the temporary object into b which is returned by create(n). So the only requirement to add in here is the copy constructor
- **c)** In line 2, the loop while(!condition_one(b)) again requires or triggers a copy contructor because the object b is passed to condition_one by value(T t) where the condition_one expects the object of type T as an argument.
- **d)** In line 3, in modify(b), the object b is passed by reference to the modify function. Since it is passed by reference, no special member fucntions are required
- **e)** In line 4, for the loop if(condition_two(b)), the object b is passed by const reference to condition_two hence no copying or assignment is needed here as well. So no special members are required
- **f)** In line 5, for return b, the object is returned from the function by value which requires a copy constructor to create a copy of b to return to the caller. Hence a copy constructor is required here

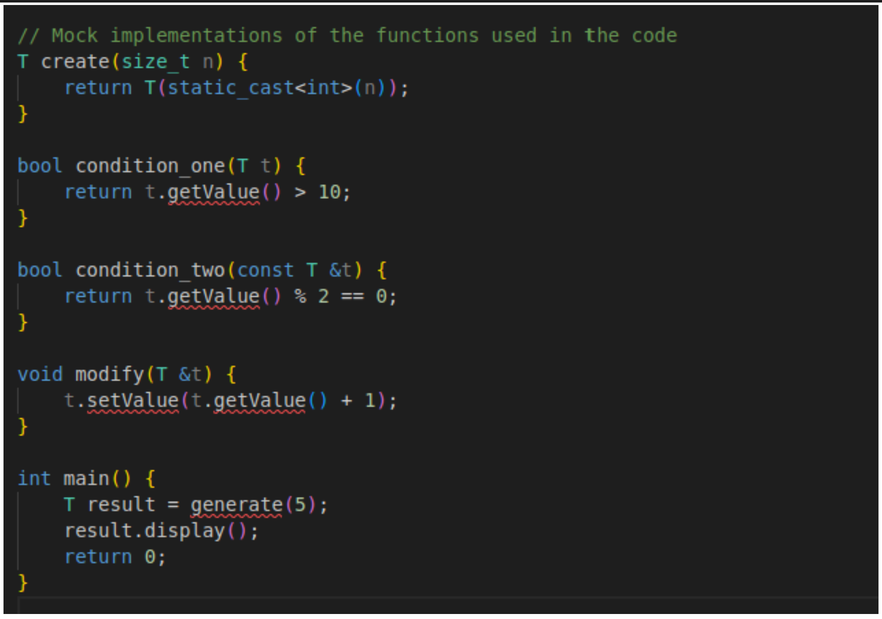
g) Summary of requirements is given in a table below

Line	Default Constructor	Copy Constructor	`operator=`
0	No	No	No
1	No	Yes	No
2	No	Yes	No
3	No	No	No
4	No	No	No
5	No	Yes	No

h) The code for the correct implementation, along with its implementation of T that satisfies the constraints

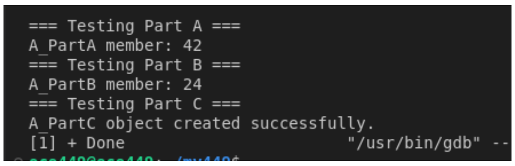


```cpp
#include <iostream>

class T {
    int value;

public:
    // Constructor accepting an integer
    T(int val = 0) : value(val) {}

    // Copy constructor
    T(const T &other) : value(other.value) {}

    // Assignment operator
    T &operator=(const T &other) {
        if (this != &other) {
            value = other.value;
        }
        return *this;
    }

    // Display function for testing
    void display() const {
        std::cout << "T value: " << value << std::endl;
    }
};

// Mock implementations of the functions used in the code
T create(size_t n) {
    return T(static_cast<int>(n));
}
```

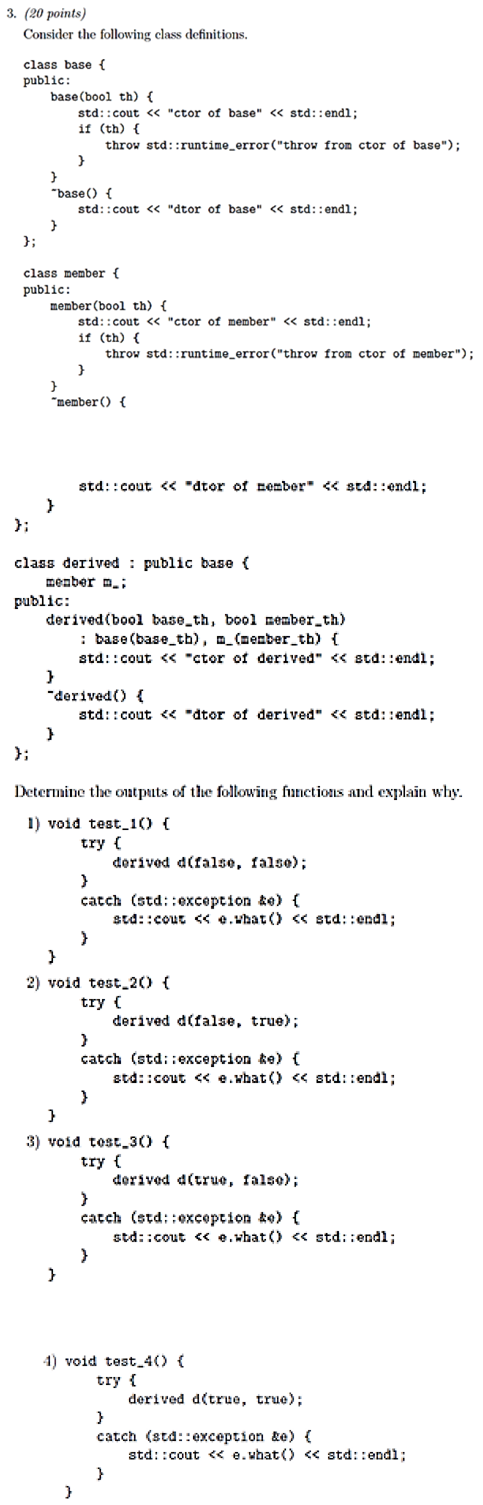


```cpp
// Mock implementations of the functions used in the code
T create(size_t n) {
    return T(static_cast<int>(n));
}

bool condition_one(T t) {
    return t.getValue() > 10;
}

bool condition_two(const T &t) {
    return t.getValue() % 2 == 0;
}

void modify(T &t) {
    t.setValue(t.getValue() + 1);
}

int main() {
    T result = generate(5);
    result.display();
    return 0;
}
```

i) The output for the code mentioned above is present here


```
=== Testing Part A ===
A_PartA member: 42
=== Testing Part B ===
A_PartB member: 24
=== Testing Part C ===
A_PartC object created successfully.
[1] + Done                       "/usr/bin/gdb" --
```

QUESTION 3:


3. *(20 points)*

Consider the following class definitions.

```
class base {
public:
    base(bool th) {
        std::cout << "ctor of base" << std::endl;
        if (th) {
            throw std::runtime_error("throw from ctor of base");
        }
    }
    ~base() {
        std::cout << "dtor of base" << std::endl;
    }
};

class member {
public:
    member(bool th) {
        std::cout << "ctor of member" << std::endl;
        if (th) {
            throw std::runtime_error("throw from ctor of member");
        }
    }
    ~member() {


        std::cout << "dtor of member" << std::endl;
    }
};

class derived : public base {
    member m_;
public:
    derived(bool base_th, bool member_th)
        : base(base_th), m_(member_th) {
        std::cout << "ctor of derived" << std::endl;
    }
    ~derived() {
        std::cout << "dtor of derived" << std::endl;
    }
};
```

Determine the outputs of the following functions and explain why.

```
1) void test_1() {
       try {
           derived d(false, false);
       }
       catch (std::exception &e) {
           std::cout << e.what() << std::endl;
       }
   }
2) void test_2() {
       try {
           derived d(false, true);
       }
       catch (std::exception &e) {
           std::cout << e.what() << std::endl;
       }
   }
3) void test_3() {
       try {
           derived d(true, false);
       }
       catch (std::exception &e) {
           std::cout << e.what() << std::endl;
       }
   }


4) void test_4() {
       try {
           derived d(true, true);
       }
       catch (std::exception &e) {
           std::cout << e.what() << std::endl;
       }
   }
```

SOLUTION: To determine the outputs of the provided functions, we analyze the given class definitions and the order in which constructors and destructors are called, along with any exceptions thrown during the process. Here is the breakdown:

Class Behavior Overview:

a) **Class base:**

I. **Constructor base(bool th):**

- Prints "ctor of base".
- Throws an exception if th is true.

II. **Destructor ~base():**

- Prints "dtor of base".

b) **Class member:**

I. **Constructor member(bool th):**

- Prints "ctor of member".
- Throws an exception if th is true.

II. **Destructor ~member():**

- Prints "dtor of member".

c) **Class derived:**

I. Inherits from base and contains a member of type member.

II. Constructor derived(bool base_th, bool member_th):

- Calls the constructor of base with base_th.
- Initializes the member m_ with member_th.
- Prints "ctor of derived".

III. **Destructor ~derived():**

- Prints "dtor of derived".

IV. **Key Points:**

- If an exception is thrown during the construction of a base class or member, destructors of already constructed objects are called to clean up.
- If an exception is thrown in the derived class constructor after all base classes and members are constructed, destructors for the base classes and members are invoked.

V. **Function-by-function analysis:**

I. **Void test_1()**

a) **base(false)**:

- Prints "ctor of base".

b) **member(false)**:

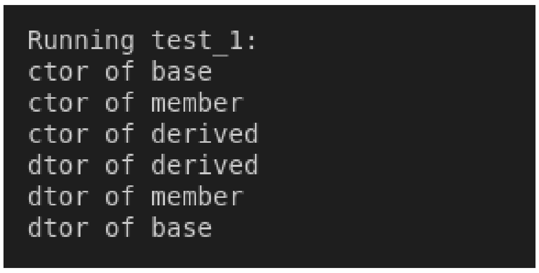
- Prints "ctor of member".

c) **derived constructor**:

- Prints "ctor of derived".

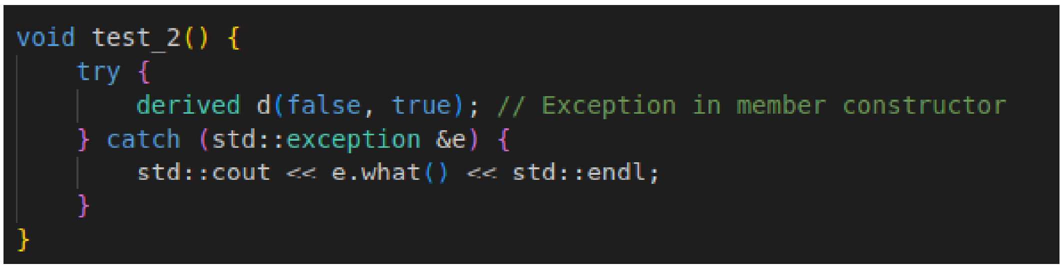
d) No exceptions are thrown, so the object d is successfully constructed.

e) Output of the function :



```
Running test_1:
ctor of base
ctor of member
ctor of derived
dtor of derived
dtor of member
dtor of base
```

II. Void test_2()



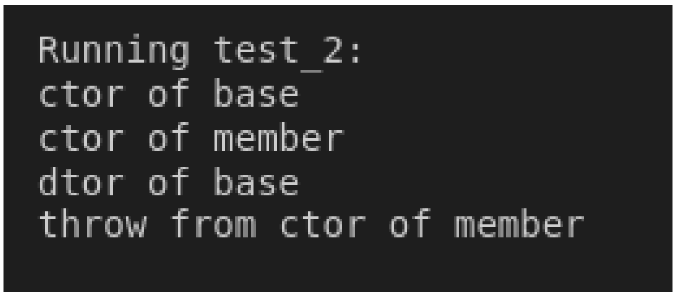
```cpp
void test_2() {
    try {
        derived d(false, true); // Exception in member constructor
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}
```

a) **base(false)**:
- Prints "ctor of base".

b) **member(true)**:
- Prints "ctor of member".
- Throws an exception: "throw from ctor of member".

c) Destructor for base is called since base was successfully constructed.
- Prints "dtor of base".

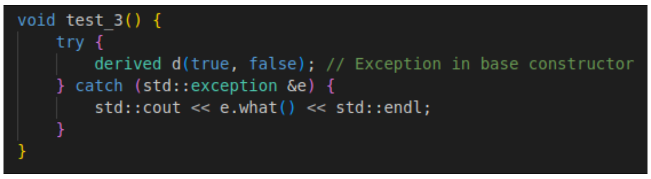
d) Output :



```
Running test_2:
ctor of base
ctor of member
dtor of base
throw from ctor of member
```

---

3) Void test_3()



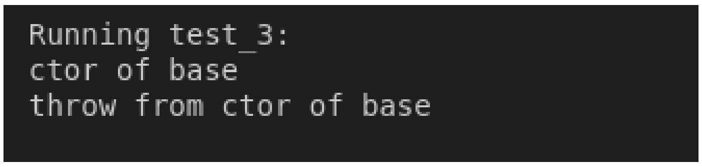
```
void test_3() {
    try {
        derived d(true, false); // Exception in base constructor
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}
```

a) **base(true)**:

- Prints "ctor of base".
- Throws an exception: "throw from ctor of base".

b) No further initialization occurs since the exception stops further construction.

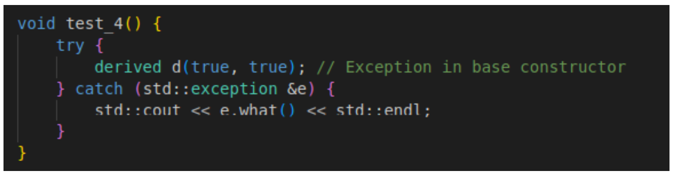
c) Output:



```
Running test_3:
ctor of base
throw from ctor of base
```

---

4) void test_4()



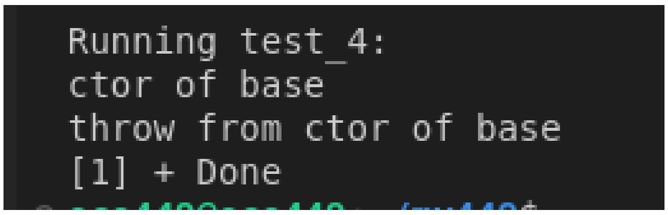
```
void test_4() {
    try {
        derived d(true, true); // Exception in base constructor
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}
```

a) **base(true)**:

- Prints "ctor of base".
- Throws an exception: "throw from ctor of base"

b) No further initialisation occurs since the exception stops further construction.

c) **Output:**



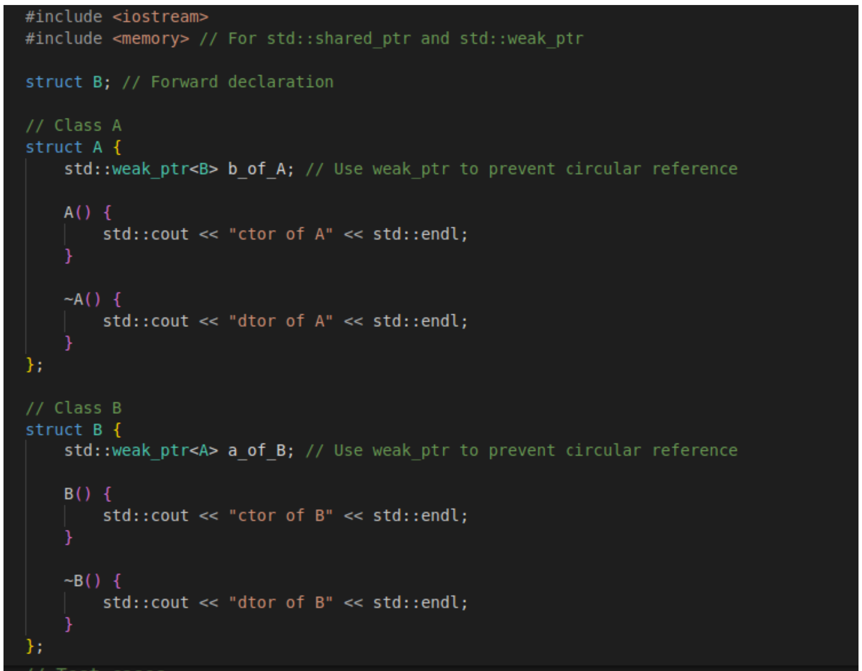
```
Running test_4:
ctor of base
throw from ctor of base
[1] + Done
```

QUESTION 4:

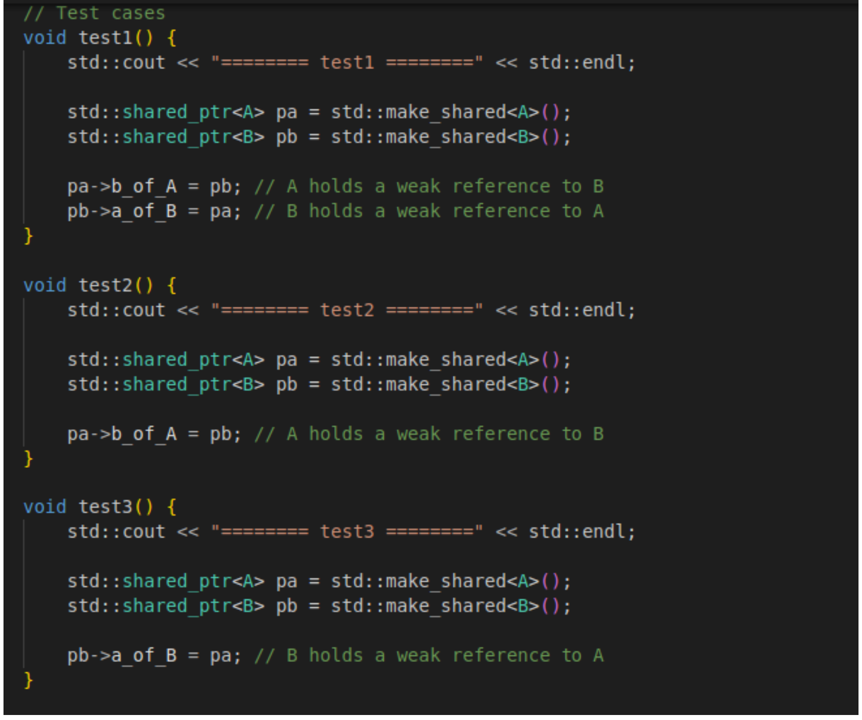
4. *(25 points)* Compile and execute `smart_pointer.cpp`. Explain the output.

SOLUTION:

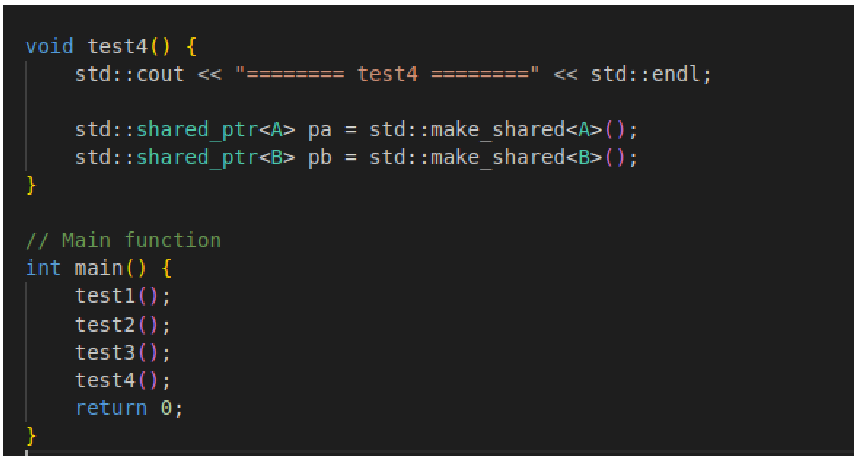
- In test1, the circular reference causes a memory leak since neither A nor B is destructed.
- This could be resolved by introducing weak_ptr to break the circular dependency. weak_ptr does not contribute to the reference count and can be used to safely reference objects managed by shared_ptr.
- The modified code version that resolves the circular reference issue is given below along with its output and discussion on the output



```cpp
#include <iostream>
#include <memory> // For std::shared_ptr and std::weak_ptr

struct B; // Forward declaration

// Class A
struct A {
    std::weak_ptr<B> b_of_A; // Use weak_ptr to prevent circular reference

    A() {
        std::cout << "ctor of A" << std::endl;
    }

    ~A() {
        std::cout << "dtor of A" << std::endl;
    }
};

// Class B
struct B {
    std::weak_ptr<A> a_of_B; // Use weak_ptr to prevent circular reference

    B() {
        std::cout << "ctor of B" << std::endl;
    }

    ~B() {
        std::cout << "dtor of B" << std::endl;
    }
};
```

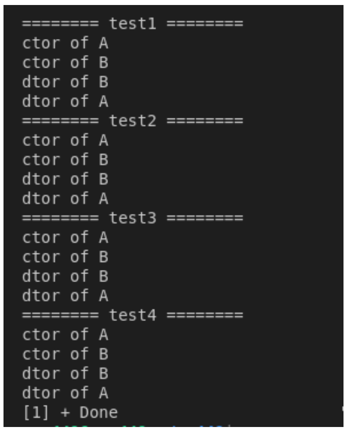


```cpp
// Test cases
void test1() {
    std::cout << "======== test1 ========" << std::endl;

    std::shared_ptr<A> pa = std::make_shared<A>();
    std::shared_ptr<B> pb = std::make_shared<B>();

    pa->b_of_A = pb; // A holds a weak reference to B
    pb->a_of_B = pa; // B holds a weak reference to A
}

void test2() {
    std::cout << "======== test2 ========" << std::endl;

    std::shared_ptr<A> pa = std::make_shared<A>();
    std::shared_ptr<B> pb = std::make_shared<B>();

    pa->b_of_A = pb; // A holds a weak reference to B
}

void test3() {
    std::cout << "======== test3 ========" << std::endl;

    std::shared_ptr<A> pa = std::make_shared<A>();
    std::shared_ptr<B> pb = std::make_shared<B>();

    pb->a_of_B = pa; // B holds a weak reference to A
}
```



```cpp
void test4() {
    std::cout << "======== test4 ========" << std::endl;

    std::shared_ptr<A> pa = std::make_shared<A>();
    std::shared_ptr<B> pb = std::make_shared<B>();
}

// Main function
int main() {
    test1();
    test2();
    test3();
    test4();
    return 0;
}
```

OUTPUT:



```
======== test1 ========
ctor of A
ctor of B
dtor of B
dtor of A
======== test2 ========
ctor of A
ctor of B
dtor of B
dtor of A
======== test3 ========
ctor of A
ctor of B
dtor of B
dtor of A
======== test4 ========
ctor of A
ctor of B
dtor of B
dtor of A
[1]  + Done
```

1. **Circular Reference Resolution:**
    - In test1, the circular reference between A and B is broken using weak_ptr. This ensures that the objects are destructed when the shared_ptr objects managing them go out of scope.
    - Without weak_ptr, the reference count of both objects would never reach zero, leading to a memory leak.
2. **Destruction Order:**
    - The order of destruction matches the reverse order of construction, which is the expected behaviour in C++ when using RAII (Resource Acquisition Is Initialization).
3. **Reliability of weak_ptr:**
    - By using weak_ptr, we avoid ownership conflicts and memory leaks, making the code robust and suitable for managing interdependent objects.

This output demonstrates the correct functioning of the modified shared_ptr and weak_ptr implementation. Circular references are effectively resolved in test1, and the other tests show expected destruction behavior for non-circular and partial links. The use of weak_ptr is crucial in preventing memory leaks in object hierarchies with bidirectional dependencies.