Programming in Modern C++: Assignment Week 12

Total Marks : 20

Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology
Kharagpur – 721302
partha.p.das@gmail.com

October 6, 2023

Question 1

Which of the following types of smart pointers follow/s exclusive ownership policy? *[MSQ, Marks 2]*

a) `std::auto_ptr`

b) `std::unique_ptr`

c) `std::std::shared_ptr`

d) `std::weak_ptr`

Answer: a), b)
Explanation:
`std::auto_ptr` (C++03) and `std::unique_ptr` (C++11) are the two smart pointer types that support **exclusive ownership policy**.
`std::std::shared_ptr` (C++11) and `std::weak_ptr` (C++11) are the two smart pointer types that support **shared ownership policy**.

Question 2

Consider the program (in C++11) given below. *[MSQ, Marks 2]*

```
#include <iostream>
#include <thread>
#include <list>
#include <functional>

struct Stat {
    std::list<int>& iLst;
    int* sum;
    int* avg;
    Stat(std::list<int>& il, int* s, int* a) : iLst{il}, sum(s), avg(a) { }
    void operator()() {
        int j = 0;
        for(auto& i : iLst) {
            *sum += i;
            i = ++j;
        }
        *avg = *sum / j;
    }
};

void show(const std::list<int>& li){
    for(auto i : li)
        std::cout << i << " ";
}

int main() {
    std::list<int> il { 10, 20, 30 };
    int s = 0, a = 0;
    std::thread t { ____________________ };    //LINE-1
    t.join();
    show(il);
    std::cout << s << " " << a;
    return 0;
}
```

Fill in the blank at `LINE-1` with appropriate option(s) such that the output becomes `1 2 3 60 20`.

a) `Stat(std::ref(il), std::ref(s), std::ref(a))`

b) `Stat(il, &s, &a)`

c) `Stat(std::ref(il), &s, &a)`

d) `std::bind(Stat, std::ref(il), s, a)`

Answer: b), c)
Explanation:
The constructor of `Stat` receives `li` as pass-by-reference, and `s` and `a` as pass-by-address. Therefore, the options b) and c) are correct. Note that `std::ref` function also enables pass-by-reference.

Question 3

Consider the code segment (C++11) given below. *[MCQ, Marks 2]*

```
#include<iostream>

template<typename T>
class SmartPtr {
    public:
        explicit SmartPtr(T* pointee): pointee_(pointee) { }
        ~SmartPtr() { delete pointee_; }
        _____________________ { return pointee_; }    //LINE-1
    private:
        T* pointee_;
};

void incr(char* p) { ++*p; }

int main(){
    SmartPtr<char> p(new char('A'));
    std::cout << *p << " ";   //LINE-1
    incr(p);
    std::cout << *p;          //LINE-2
    return 0;
}
```

Fill in the blank at `LINE-1` with appropriate option(s) such that the output becomes `A B`.

a) `T& operator*() const`

b) `operator T*()`

c) `T* operator->() const`

d) `operator void*()`

Answer: b)
Explanation:
Since `LINE-1` and `LINE-2` need to print `SmartPtr` as `char`, we have to implement the corresponding typecasting operator. Therefore, option b) is the correct option.

Question 4

Consider the code segment (C++11) given below. *[MCQ, Marks 2]*

```
#include<iostream>

template<typename T>
class SmartPtr {
    public:
        explicit SmartPtr(T* pointee): pointee_(pointee) { }
        ~SmartPtr() { delete pointee_; }
        T& operator*() const { return *pointee_; }
        T* operator->() const { return pointee_; }
        operator T*() { return pointee_; }
        operator void*() { return pointee_; }
    private:
        T* pointee_;
};

void incr(char* p) { ++*p; }

int main(){
    SmartPtr<char> p(new char('A'));
    std::cout << *p << " ";    //LINE-1
    incr(p);
    std::cout << *p;           //LINE-2
    delete p;                  //LINE-3
    return 0;
}
```

What will be the output/error?

a) `A B`

b) `A A`

c) `compiler error at LINE-1 and LINE-2: cannot convert 'SmartPtr' to 'char*'`

d) `compiler error at LINE-3: ambiguous default type conversion from 'SmartPtr'`

Answer: d)
Explanation:
The statement `delete p;` generates a compiler error since the compiler cannot decide which conversion (either to `char*` or to `void*`) to apply. Thus, it results in `ambiguous default type conversion from 'SmartPtr'`.

Question 5

Consider the following code segment (int C++11). *[MCQ, Marks 2]*

```
#include <iostream>
#include <functional>

int fun(int i, int j, int k, const int& l) {
    return (i + j) - (k + l);
}
int main() {
    using namespace std::placeholders;
    int a = 10, b = 20;
    auto wf = std::bind(fun, _3, a, _1, std::cref(b));
    std::cout << wf(5, 6, 7) << " ";
    a = b = -10;
    std::cout << wf(5, 6, 7);
    return 0;
}
```

What will be the output?

a) `-8 -8`

b) `-11 19`

c) `-12 18`

d) `-8 22`

Answer: d)
Explanation:
The call `wf(5, 6, 7)` results in binding 5 to _1, 6 to _2 (which is not used), and 3 to _3. The formal arguments for the first call to `compute` are as `fun(7, 10, 5, 20)`, which is evaluated as `-8`. The formal arguments for the second call to `compute` are as `print(7, 10, 5, -10)` (since `b` is considered as reference type in `bind` function), which is evaluated as `22`.

Question 6

Consider the code segment (C++11) given below. *[MCQ, Marks 2]*

```
#include<iostream>
#include <memory>

void f(std::shared_ptr<char> cp){
    std::shared_ptr<char> cp1(cp);
    std::cout << "rc = " << cp1.use_count() << " ";
}
int main(){
    std::shared_ptr<char> cp1 = std::make_shared<char>('A');
    {
        std::shared_ptr<char> cp2(cp1);
        std::cout << "rc = " << cp1.use_count() << " ";
    }
    std::shared_ptr<char> cp3(cp1);
    f(cp3);
    std::cout << "rc = " << cp1.use_count() << " ";
    cp3.reset(new char('B'));
    std::cout << "rc = " << cp1.use_count();
    return 0;
}
```

What will be the output?

a) `rc = 2 rc = 3 rc = 3 rc = 1`

b) `rc = 2 rc = 3 rc = 3 rc = 2`

c) `rc = 2 rc = 4 rc = 2 rc = 1`

d) `rc = 2 rc = 4 rc = 2 rc = 2`

Answer: c)
Explanation:
The code is explained in the comment:

```
#include<iostream>
#include <memory>

void f(std::shared_ptr<char> cp){                              //rc = 3
    std::shared_ptr<char> cp1(cp);                             //rc = 4
    std::cout << "rc = " << cp1.use_count() << " ";
}                                                              //local cp1 get deleted
                                                                   on return
int main(){
    std::shared_ptr<char> cp1 = std::make_shared<char>('A');   //rc = 1
    {
        std::shared_ptr<char> cp2(cp1);                        //rc = 2
        std::cout << "rc = " << cp1.use_count() << " ";
    }                                                          //rc = 1
    std::shared_ptr<char> cp3(cp1);                            //rc = 2
    f(cp3);
```

```
    std::cout << "rc = " << cp1.use_count() << " ";              //rc = 3
    cp3.reset(new char('B'));                                    //rc = 1
    std::cout << "rc = " << cp1.use_count();
    return 0;
}
```

Question 7

Consider the code segment (C++11) given below. *[MSQ, Marks 2]*

```
#include <iostream>
#include <list>
#include <thread>

void addToList(std::list<char>& lc){
    lc.push_back('C');
}

int main(){
    std::list<char> lc;
    for(int i = 0; i < 2; i++)
        lc.push_back('A' + i);
    std::thread t1 {std::ref(addToList), std::ref(lc)};
    t1.join();
    for(char c : lc)
        std::cout << c << " ";
    return 0;
}
```

What is/are the possible output?

a) `A C B`

b) `A B C`

c) `C A B`

d) It prints nothing

Answer: b)
Explanation:
The `main` adds 'A' and 'B' to the `std::list<char> lc`, and then create the thread. The execution of the thread `t1` must be completed before (due to `join` call) printing the list in `main`. Thus, it always prints `A B C`.
Intentionally kept as MSQ.

Question 8

Consider the following program (in C++11). *[MSQ, Marks 2]*

```
#include <iostream>
#include <functional>
#include <thread>
#include <mutex>

struct ResourceA{
    int RAC;
};

struct ResourceB{
    int RBC;
};

std::mutex RA_mtx;
std::mutex RB_mtx;

void req1(ResourceA& obj_A, ResourceB& obj_B, int nA, int nB) {
    std::unique_lock<std::mutex> lck1(RA_mtx);
    std::unique_lock<std::mutex> lck2(RB_mtx);
    obj_A.RAC += nA;
    obj_B.RBC += nB;
    std::cout << "REQ1: " << obj_A.RAC << " " << obj_B.RBC << std::endl;
}

void req2(ResourceA& obj_A, ResourceB& obj_B, int nA, int nB) {
    std::unique_lock<std::mutex> lck2(RB_mtx);
    std::unique_lock<std::mutex> lck1(RA_mtx);
    obj_A.RAC += nA;
    obj_B.RBC += nB;
    std::cout << "REQ2: " << obj_A.RAC << " " << obj_B.RBC << std::endl;
}

int main(){
    ResourceA rA{0};
    ResourceB rB{0};
    std::thread t1{ std::bind(req1, std::ref(rA), std::ref(rB), 5, 5) };
    std::thread t2{ std::bind(req2, std::ref(rA), std::ref(rB), 4, 4) };
    t1.join();
    t2.join();
    return 0;
}
```

Identify the statement/s that is/are **not true** about the program.

a) It generates output as:
```
REQ2:  4 4
REQ1:  9 9
```

b) It generates output as:
```
REQ1:  5 5
REQ2:  4 4
```

c) It generates output as:
```
REQ1:  5 5
REQ2:  9 9
```

d) It results in deadlock

Answer: b)
Explanation:
Since the code in `req1` and `req2` execute in a mutual exclusive manner, the output can be:
```
REQ1:  5 5
REQ2:  9 9
```
or
```
REQ2:  4 4
REQ1:  9 9
```
However, it cannot be
```
REQ1:  5 5
REQ2:  4 4
```
It may also happen that `t1` holds lock on `RA_mtx`, and `t2` holds lock on `RB_mtx`. Then, `t1` request to lock on `RB_mtx`, and `t2` requests lock on `RA_mtx`. It results in a deadlock. Therefore, b) is the correct option.
Intentionally kept as MSQ

Question 9

Consider the following code segment (in C++11). *[MCQ, Marks 2]*

```
#include <iostream>
#include <future>
#include <list>

struct Prod{
    Prod(const std::list<int>& dl) : dl_(dl) { }
    double operator()() {
        int p = 1;
        for(int it : dl_)
            p *= it;
        return p;
    }
    std::list<int> dl_;
};

double callProd(const std::list<int>& dl){
    auto as = _______________________________;    //LINE-1
    return as.get() ;
}

int main() {
    std::list<int> dLi {2, 4, 6, 2, 5};
    std::cout << callProd(dLi);
    return 0;
}
```

Choose the appropriate option to fill in the blank at `LINE-1` such that output becomes `480`.

a) `std::thread(Prod(dl))`

b) `std::thread{std::bind(Prod(dl))}`

c) `std::async(Prod(dl))`

d) `std::atomic(std::ref(Prod(dl)))`

Answer: c)
Explanation:
Since `as.get()` must be waiting for fullfillment of the promise, which the return value of `Prod`, the call at `LINE-1` must be `std::async(Prod(dl))`.

Programming Questions

Question 1

Consider the following program (in C++11).

- Fill in the blank at `LINE-1` with appropriate header to overload function operator.
- Fill the blank at `LINE-2` to invoke the functor `Factorial()` asynchronously.
- Fill the blank at `LINE-3` to receive the output from functor `Factorial()`.

The program must satisfy the sample input and output. *Marks: 3*

```
#include <iostream>
#include <future>

struct Factorial{
    Factorial(const long long& n) : n_( n) { }
    ______________________________________ {        //LINE-1
        long long f = 1;
        if(n_ == 0 || n_ == 1)
            return 1;
        for(int i = 1; i <= n_; i++)
            f *= i;
        return f;
    }
    const long long n_;
};

long long callFacto(int n){
    auto a = __________________________;            //LINE-2
    return ______________________;                  //LINE-3
}

int main() {
    int n;
    std::cin >> n;
    std::cout << callFacto(n);
    return 0;
}
```

Public 1

```
Input: 10
Output: 3628800
```

Public 2

```
Input: 3
Output: 6
```

Private

```
Input: 11
Output: 39916800
```

Answer:
`LINE-1:  long long operator()()`
`LINE-2:  std::async(Factorial(n))`
`LINE-3:  a.get()`
Explanation:
The function header at `LINE-1` to overload the function operator can be: `long long operator()()`
At `LINE-2` the asynchronous call to the functor `Factorial()` can be made as:
`auto a = std::async(Factorial(n))`
At `LINE-3` can use the statement `a.get()` to receive the result of functor `Factorial()`.

Question 2

Consider the following program (in C++11).

- Fill in the blank at `LINE-1` by defining a mutex object.
- Fill the blanks at `LINE-2` and `LINE-4` by locking the mutex object.
- Fill the blanks at `LINE-3` and `LINE-5` by unlocking the mutex object.

The program must satisfy the sample input and output. *Marks: 3*

```
#include <iostream>
#include <thread>
#include <functional>
#include <chrono>
#include <mutex>

_________________________;   //LINE-1

class muffin_store {
    public:
        muffin_store() : n_muffins(0) {};
        void in_stock(int m){
            ________________________;    //LINE-2
            update_amount = m;
            int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 20);
            std::this_thread::sleep_for(std::chrono::milliseconds(delay));
            n_muffins += update_amount;
            ________________________;    //LINE-3
        }
        void out_stock(int m){
            ________________________;    //LINE-4
            update_amount = m;
            int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 40);
            std::this_thread::sleep_for(std::chrono::milliseconds(delay));
            n_muffins -= update_amount;
            ________________________;    //LINE-5
        }
        void show_stock() { std::cout << n_muffins; }
    private:
        int n_muffins;
        int update_amount;
};

void incoming_muffin(muffin_store& ms, int n){
    for(int i = 1; i <= n; i++){
        int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 10);
        std::this_thread::sleep_for(std::chrono::milliseconds(delay));
        ms.in_stock(i * 10);
    }
}

void outgoing_muffin(muffin_store& ms, int n){
```

```
    for(int i = n; i >= 1; i--){
        int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 30);
        std::this_thread::sleep_for(std::chrono::milliseconds(delay));
        ms.out_stock(i * 10);
    }
}

int main(){
    int n, m;
    std::cin >> n >> m;
    muffin_store ms;
    std::thread t1{ std::bind(incoming_muffin, std::ref(ms), n) };
    std::thread t2{ std::bind(outgoing_muffin, std::ref(ms), m) };

    t1.join();
    t2.join();
    ms.show_stock();
    return 0;
}
```

Public 1

Input: 10 10
Output: 0

Public 2

Input: 20 10
Output: 1550

Private

Input: 10 20
Output: -1550

Answer:
LINE-1: `std::mutex mf_mtx;`
LINE-2: `mf_mtx.lock()`
LINE-3: `mf_mtx.unlock()`
LINE-4: `mf_mtx.lock()`
LINE-5: `mf_mtx.unlock()`
Explanation:
At LINE-1, the mutex object can be created as:
`std::mutex ac_mtx;`
At LINE-2 and LINE-4, locking of the mutex object can be written as:
`ac_mtx.lock()`
At LINE-3 and LINE-5, unlocking of the mutex object can be written as:
`ac_mtx.unlock()`

Question 3

Consider the following program in C++14 to represent a generic deque (double-ended-queue), which allows adding items at the front of the queue and at the end of the queue. Complete the program as per the instructions given below.

- Fill in the blank at `LINE-1` with appropriate statements so that after execution of the `for` loop, `t` refers to the last element of the deque.
- Fill in the blank at `LINE-2` with appropriate statements to traverse the list in forward direction.
- fill in the blank at `LINE-3` with appropriate statements to traverse the list in backward direction,

The program must satisfy the sample input and output. *Marks: 3*

```
#include<iostream>
#include<memory>

namespace DS{
    template<typename T1>
    class deque;

    template<typename T2>
    class node{
        public:
            node(T2 _info) : info(_info), next(nullptr) {}
            friend deque<T2>;
        private:
            T2 info;
            std::shared_ptr<node<T2>> next;
            std::weak_ptr<node<T2>> prev;
    };

    template<typename T1>
    class deque{
        public:
            deque() = default;
            void addFront(const T1& item){
                std::shared_ptr<node<T1>> n = std::make_shared<node<T1>>(item);
                if(first == nullptr){
                    first = n;
                    last = first;
                }
                else{
                    n->next = first;
                    first->prev = n;
                    first = n;
                }
            }
            void addEnd(const T1& item){
                std::shared_ptr<node<T1>> n = std::make_shared<node<T1>>(item);
                if(first == nullptr){
```

```
                    first = n;
                    last = first;
                }
                else{
                    std::shared_ptr<node<T1>> t = first;
                    for(___________________________________________);    //LINE-1
                    t->next = n;
                    n->prev = t;
                    last = n;
                }
            }
            void traverse(){
                for(_______________________________________________)     //LINE-1
                    std::cout << t->info << " ";
            }
            void rev_traverse(){
                for(______________________________________________)      //LINE-2
                    std::cout << p->info << " ";
            }
            private:
                std::shared_ptr<node<T1>> first { nullptr };
                std::shared_ptr<node<T1>> last  { nullptr };
    };
}

int main(){
    DS::deque<int> il;
    int n, a;
    std::cin >> n;
    for(int i = 0; i < n; i++){
        std::cin >> a;
        il.addFront(a);
    }
    for(int i = 0; i < n; i++){
        std::cin >> a;
        il.addEnd(a);
    }
    il.traverse();
    std::cout << std::endl;
    il.rev_traverse();
    return 0;
}
```

Public 1

```
Input:
4
10 20 30 40
50 60 70 80
Output:
40 30 20 10 50 60 70 80
80 70 60 50 10 20 30 40
```

Public 2

```
Input:
3
10 20 30
40 50 60
Output:
30 20 10 40 50 60
60 50 40 10 20 30
```

Public 3

```
Input:
2
8 9
7 5
Output:
9 8 7 5
5 7 8 9
```

Private

```
Input:
5
1 2 3 4 5
6 7 8 9 10
Output:
5 4 3 2 1 6 7 8 9 10
10 9 8 7 6 1 2 3 4 5
```

Answer:

```
LINE-1:  ; t->next != nullptr; t = t->next
LINE-2:  std::shared_ptr<node<T1>> t = first; t != nullptr; t = t->next
LINE-3:  std::weak_ptr<node<T1>> t = last; auto p = t.lock(); t = p->prev
```

Explanation:
To add a node at the end of the deque, we can find that t is already initialized to the first element of the deque. Therefore, the for loop can be written as follows:

```
std::shared_ptr<node<T1>> t = first;
for(; t->next != nullptr; t = t->next);
```

The ; at the end of for loop indicate an empty statement. Since in class node, next is a shared_ptr, the for loop for forward traversal must be:

```
std::shared_ptr<node<T>> t = first; t != nullptr; t = t->next)
```

Since in class node, prev is a weak_ptr, the for loop for reverse traversal must be:

```
std::weak_ptr<node<T>> t = last; auto p = t.lock(); t = p->prev
```

Note that last is a shared_ptr while prev is a weak_ptr. So we need to devise a way to navigate between the two. Recall that weak_ptr cannot be used to access the the pointee. So we first get weak_ptr t from last. Now for access, we get a shared_ptr p by locking the weak_ptr t. This will used in the loop body. Finally, to progress backward, we get p->prev and keep in the weak_ptr t. That completes the solution.