Programming in Modern C++: Assignment Week 12

Total Marks : 27

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

April 16, 2025

Question 1

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

```
#include <iostream>
#include <thread>
#include <list>
using namespace std;

struct Product {
    list<int>& iLst;
    int* result;
    Product(list<int>& il, int* res) : iLst{il}, result{res} { }
    void operator()() {
        for(auto& i : iLst) {
            *result *= i;
            i = 0;
        }
    }
};

void print(list<int> iLst) {
    for(auto i : iLst)
        std::cout << i << " ";
}
int main() {
    list<int> il { 10, 20, 30 };
    int res = 1;
    std::thread t { ___________________ };    //LINE-1
    t.join();
    print(il);
    std::cout << res;
    return 0;
}
```

Fill in the blank at `LINE-1` with appropriate option(s) such that the output becomes `0 0 0 6000`.

a) `Product(&il, &res)`

b) `Product(il, &res)`

c) `Product(std::ref(il), &res)`

d) `std::bind(Product, std::ref(il), &res)`

Answer: b), c)
Explanation:
The constructor of `Product` receives `li` as pass-by-reference and `res` as pass-by-address.
At option a), the actual parameter `&li` is an address. Therefore, it is a wrong option.
At option b), the actual parameter `li` is a correct way to perform pass-by-reference (or pass-by-value) and `&res` is passed-by-address. Therefore, it is a correct option.
At option c), the actual parameter `std::ref(li)` is a pass-by-reference explicitly and `&res` is passed-by-address. Therefore, it is a correct option.
At option d), the actual parameter `Product` of `std::bind` function is a call to the default constructor of functor `Product`, which is not provided. Therefore, it is a wrong option.

Question 2

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

```
#include <iostream>
#include <functional>
using namespace std;

int print(int n1, int n2, int n3, const int& n4) {
    return (n1 - (n2 + n3 + n4));
}
int main() {
    using namespace std::placeholders;
    int val = 10;
    auto f = std::bind(print, _2, val, _1, std::cref(val));
    val = -10;
    std::cout << f(100, 20, -100);
    return 0;
}
```

What will be the output?

a) `-100`

b) `-80`

c) `80`

d) `120`

Answer: b)
Explanation:
The call `f(100, 20, -100)` results in binding 100 to _1, 20 to _2, and -100 remains unused. The formal arguments to the call to `print` are as `print(20, 10, 100, -10)`, which is evaluated as `-80`.

Question 3

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

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

std::mutex val_mutex;
int val = 1;

void stepUp(const int& steps){
    std::unique_lock<std::mutex> lck(val_mutex);
    int i = val;
    i += steps;
    val = i;
}

int main() {
    std::thread t1 { &stepUp, 1 };
    std::thread t2 { &stepUp, 2 };
    std::thread t3 { &stepUp, 3 };
    t1.join();
    t2.join();
    t3.join();
    std::cout << val;
    return 0;
}
```

What will be the output?

a) `1`

b) `6`

c) `7`

d) Unpredictable

Answer: c)
Explanation:
Since within function `stepUp` all the modifications to `val` takes place in a mutual exclusive way, each thread will update the value of `val` and the result will be `7`. However, the order in which order threads update `val` cannot be predicted.

Question 4

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

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

std::mutex m;
thread_local int val = 1;
void stepUp(const int& steps){
    m.lock();
    int i = val;
    i += steps;
    val = i;
    m.unlock();
}

int main() {
    val = 0;
    std::thread t1 { std::bind(stepUp, 1) };
    std::thread t2 { std::bind(stepUp, 2) };
    std::thread t3 { std::bind(stepUp, 3) };
    t1.join();
    t2.join();
    t3.join();
    std::cout << val;
    return 0;
}
```

What will be the output?

a) `0`

b) `1`

c) `6`

d) `7`

Answer: a)
Explanation:
Since the definition `thread_local int val = 1;` makes `val` local to each of the threads, each thread increments its local copy of `val`. Thus, output becomes `100` in `main`.

Question 5

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

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

struct Printer{
    int pCounter = 0;
};

struct Scanner{
    int sCounter = 0;
};

std::mutex printer_mutex;
std::mutex scanner_mutex;
Printer printer;
Scanner scanner;

void requestCopy1(int n) {
    std::unique_lock<std::mutex> lck1(printer_mutex);
    std::unique_lock<std::mutex> lck2(scanner_mutex);
    printer.pCounter += n;
    scanner.sCounter += n;
    std::cout << printer.pCounter << ", " << scanner.sCounter << std::endl;
}

void requestCopy2(int n) {
    std::unique_lock<std::mutex> lck1(scanner_mutex);
    std::unique_lock<std::mutex> lck2(printer_mutex);
    printer.pCounter += n;
    scanner.sCounter += n;
    std::cout << printer.pCounter << ", " << scanner.sCounter << std::endl;
}

int main(){
    std::thread t1(requestCopy1, 10);
    std::thread t2(requestCopy2, 20);
    t1.join();
    t2.join();
    return 0;
}
```

Which of the following cannot be true about the output of the above program?

a) It generates output as:
```
10, 10
30, 30
```

b) It generates output as:
```
20, 20
30, 30
```

c) It generates output as:
`20, 20`
`10, 10`

d) It results in deadlock

Answer: a), b)
Explanation:
Since the code in `requestCopy1` and `requestCopy2` execute in a mutual exclusive manner, the output can be:
`10, 10`
`30, 30`
or
`20, 20`
`30, 30`
However, it may happen that `t1` holds lock on `printer_mutex`, and `t2` holds lock on `scanner_mutex`. Then, `t1` request to lock on `scanner_mutex`, and `t2` requests lock on `scanner_mutex`. It results in a deadlock.

Question 6

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

```
#include <iostream>
#include <vector>
#include <thread>

void add_to_vec(std::vector<int>& iVec){
    for(int i = 3; i < 5; i++)
        iVec.push_back(i);
}

int main(){
    std::vector<int> vc;
    for(int i = 1; i < 3; i++)
        vc.push_back(i);
    std::thread t1 {add_to_vec, std::ref(vc)};
    for(auto i : vc)
        std::cout << i << " ";
    t1.join();
    return 0;
}
```

Which of the following **cannot** be valid output?

a) `1 2`

b) `1 2 3 4`

c) `1 3 2 4`

d) `3 4 1 2`

Answer: c), d)
Explanation:
The `main` thread adds `1` and `2` to vector `vc` before the thread `t` got initiated. The thread `t` and the `for` loop printing `vc` work in an interleaved manner. Thus, the options c) and d) are not possible.

Question 7

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

```
#include <iostream>

template <class T>
class SmartPtr {
    private:
        T* _pointee;
    public:
        explicit SmartPtr(T* p = nullptr) { _pointee = p; }
        ~SmartPtr() { delete _pointee; }
        ________________________ { return *_pointee; }    //LINE-1
};

int main() {
    const SmartPtr<char> sp(new char);
    *sp = 'X';
    std::cout << *sp;
    return 0;
}
```

Identify the correct function header at `LINE-1` so that it prints **X**.

a) `T operator* () const`

b) `T& operator* () const`

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

d) `SmartPtr& operator=`

Answer: b)
Explanation:
The statements `*sp = 'X';` as well as `std::cout << *sp;` require dereferencing operator to be overridden. Thus, option b) is the correct option.

Question 8

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

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

void update_share_ptr(const std::shared_ptr<int>& sp){
    std::shared_ptr<int> lp = sp;
    std::cout << "RC = " << lp.use_count() << " ";
}
int main(){
    std::shared_ptr<int> spA(new int(10));
    {
        std::shared_ptr<int> spB(spA);
        std::cout << "RC = " << spA.use_count() << " ";
    }
    std::shared_ptr<int> spC = spA;
    std::cout << "RC = " << spA.use_count() << " ";
    update_share_ptr(spA);
    std::cout << "RC = " << spA.use_count() << " ";
    spC.reset(new int(20));
    std::cout << "RC = " << spA.use_count() << " ";
    return 0;
}
```

What will be the output?

a) `RC = 1 RC = 1 RC = 2 RC = 2 RC = 1`

b) `RC = 2 RC = 2 RC = 4 RC = 2 RC = 1`

c) `RC = 2 RC = 2 RC = 4 RC = 3 RC = 2`

d) `RC = 2 RC = 2 RC = 3 RC = 2 RC = 1`

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

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

void update_share_ptr(const std::shared_ptr<int>& sp){ //RC = 2 since pass-by-reference
    std::shared_ptr<int> lp = sp;            //RC = 3
    std::cout << "RC = " << lp.use_count() << " ";
}
int main(){
    std::shared_ptr<int> spA(new int(10));   //RC = 1
    {
        std::shared_ptr<int> spB(spA);       //RC = 2
        std::cout << "RC = " << spA.use_count() << " ";
    }
    std::shared_ptr<int> spC = spA;          //RC = 2 since spB got deleted
    std::cout << "RC = " << spA.use_count() << " ";
```

```
    update_share_ptr(spA);                    //call to update_share_ptr
    std::cout << "RC = " << spA.use_count() << " ";
    spC.reset(new int(20));                   //RC = 1 since local lp got deleted
    std::cout << "RC = " << spA.use_count() << " ";
    return 0;
}
```

Question 9

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

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

void update_share_ptr1(std::shared_ptr<int>& sp){
    sp.reset(new int(3));
}

void update_share_ptr2(std::shared_ptr<int> sp){
    sp.reset(new int(4));
}

int main() {
    auto sp1 = std::make_shared<int>(1);
    auto sp2 = sp1;
    std::weak_ptr<int> wp1 = sp1;
    std::weak_ptr<int> wp2 = sp2;
    update_share_ptr1(sp1);
    update_share_ptr2(sp2);

    if(auto p = wp1.lock())
        std::cout << *p << std::endl;
    else
        std::cout << "wp1 is expired" << std::endl;
    update_share_ptr1(sp2);
    update_share_ptr2(sp1);

    if(auto p = wp2.lock())
        std::cout << *p << std::endl;
    else
        std::cout << "wp2 is expired" << std::endl;
    return 0;
}
```

What will be the output?

a) 1
   2

b) `wp1 is expired`
   `1`

c) `1`
   `wp2 is expired`

d) `wp1 is expired`
   `wp2 is expired`

Answer: c)
Explanation:
Since function `update_share_ptr1` uses pass-by-refernce as parameter, it resets the actual shared pointer passed as parameter.

Since function `update_share_ptr2` uses pass-by-value as parameter, it does not affect the actual shared pointer passed as parameter.
The statement `auto sp2 = sp1;` sets RC $= 2$ for `sp1` as well as `sp2`. Weak pointers `wp1` and `wp2` refers to `sp1` and `sp2`.
The statements `update_share_ptr1(sp1);` and `update_share_ptr2(sp2);` results in `RC = 1` (since the second call does not affect the share pointers in `main`). Thus, it prints 1 as the first output.
Next, the statements `update_share_ptr2(sp1);` and `update_share_ptr1(sp2);` results in `RC = 0`. Thus, it prints `wp2 is expired`.

Programming Questions

Question 1

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

- Fill in the blank at `LINE-1` and `LINE-2` with appropriate definitions of `std::promise p1` and `std::future f1`.
- Fill in the blank at `LINE-3` and `LINE-4` with appropriate definitions of `std::promise p2` and `std::future f2`.
- Fill in the blank at `LINE-5` to define a `std::thread` object by passing appropriate `std::future` and `std::promise` objects.

The program must satisfy the given test cases. *Marks: 3*

```
#include <iostream>
#include <functional>
#include <thread>
#include <future>
#include <vector>

void product (std::promise<long>& p, std::future<std::vector<int>>& f) {
    std::vector<int> v = f.get();
    long result = 1;
    for(int i : v)
        result *= (long)i;
    p.set_value(result);
}

int main () {
    ___________________________________;    //LINE-1
    ___________________________________;    //LINE-2

    ___________________________________;    //LINE-3
    ___________________________________;    //LINE-4

    ___________________________________;    //LINE-5

    int n, a;
    std::cin >> n;
    std::vector<int> v;
    for(int i = 0; i < n; i++){
        std::cin >> a;
        v.push_back(a);
    }

    p1.set_value (v);
    std::cout << f2.get();
    th.join();
    return 0;
}
```

Public 1

```
Input: 3 10 20 30
Output: 6000
```

Public 2

```
Input: 4 10 -10 20 20
Output: -40000
```

Private

```
Input: 5 2 -3 -4 5 6
Output: 720
```

Answer:

```
LINE-1:  std::promise<std::vector<int>> p1
LINE-2:  std::future<std::vector<int>> f1 = p1.get_future()
LINE-3:  std::promise<long> p2
LINE-4:  std::future<long> f2 = p2.get_future()
LINE-5:  std::thread th (product, std::ref(p2), std::ref(f1))
```

Explanation:

At `LINE-1` and `LINE-2`, the `std::promise` `p1` and `std::future` `f1` can be created as follows:

```
LINE-1:  std::promise<std::vector<int>> p1
LINE-2:  std::future<std::vector<int>> f1 = p1.get_future()
```

At `LINE-3` and `LINE-4`, the `std::promise` `p2` and `std::future` `f2` can be created as follows:

```
LINE-3:  std::promise<long> p2;
LINE-4:  std::future<long> f2 = p2.get_future()
```

The function `product` to get the values from `main` using `std::promise` `p1`, whereas it sets the value to `std::future` `f2` so that it can be accessed by `main`. Therefore, the thread can be defined as follows

```
LINE-5:  std::thread th (product, std::ref(p2), std::ref(f1))
```

Question 2

Consider the following program (in C++14).

- Fill in the blank at `LINE-1` with an appropriate definition of mutex `mq_mutex`.
- Fill in the blanks at `LINE-2` and `LINE-3` with appropriate statements such that all the modification to `total_elems` happens in a mutual exclusive manner.

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

```
#include <iostream>
#include <vector>
#include <thread>
#include <functional>
#include <chrono>
#include <mutex>
____________________________;    //LINE-1

class MessageQueue{
    private:
        int total_elems;
        int n;
    public:
        MessageQueue() : total_elems(0) {}
        void enqueue(int num_elems){
            __________________________________;    //LINE-2
            n = num_elems;
            int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 10);
            std::this_thread::sleep_for(std::chrono::milliseconds(delay));
            total_elems += n;
        }
        void dequeue(int num_elems){
            ___________________________________;    //LINE-3
            n = num_elems;
            int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 10);
            std::this_thread::sleep_for(std::chrono::milliseconds(delay));
            total_elems -= n;
        }
        int getElemsCount(){
            return total_elems;
        }
};

void sender(MessageQueue& mq, int n){
    for(int i = 0; i < n; i++){
        int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 10);
        std::this_thread::sleep_for(std::chrono::milliseconds(delay));
        mq.enqueue(i + 1);
    }
}

void receiver(MessageQueue& mq, int n){
    for(int i = n - 1; i >= 0; i--){
```

```
        int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 30);
        std::this_thread::sleep_for(std::chrono::milliseconds(delay));
        mq.dequeue(i + 1);
    }
}

int main(){
    int n, m;
    std::cin >> n;
    std::cin >> m;
    MessageQueue mq;
    std::thread t1{ std::bind(sender, std::ref(mq), n) };
    std::thread t2{ std::bind(receiver, std::ref(mq), m) };

    t1.join();
    t2.join();
    std::cout << mq.getElemsCount();
    return 0;
}
```

Public 1

Input: 10 10
Output: 0

Public 2

Input: 20 10
Output: 155

Public 3

Input: 10 20
Output: -155

Private

Input: 20 20
Output: 0

Answer:
LINE-1: `std::mutex mq_mutex`
LINE-2: `std::unique_lock<std::mutex> lck(mq_mutex)`
LINE-3: `std::unique_lock<std::mutex> lck(mq_mutex)`
Explanation:
At `LINE-1`, mutex can be defined as:
`std::mutex mq_mutex`
At `LINE-2` and `LINE-3` we can use the following statement:
`std::unique_lock<std::mutex> lck(mq_mutex)` such that any update to the variable `total_elem` would be done in a critical section.

Question 3

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 `NthPrime()` asynchronously.
- Fill the blank at `LINE-3` to receive the output from functor `NthPrime()`.

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

```
#include <iostream>
#include <future>
#include <cmath>
using namespace std;

struct NthPrime{
    const long long n;
    NthPrime(const long long& n) : n( n) { }
    bool isPrime(long long n){
        if (n <= 1)
            return false;
        for (int i = 2; i <= sqrt(n); i++)
            if (n % i == 0)
                return false;

        return true;
    }
    ____________________________________ {     //LINE-1
        long long num = 2;
        for(int i = 0; i < n; num++) {
            if (isPrime(num)) {
                ++i;
            }
        }
        return num-1;
    }
};

long long findNthPrime(int n){
    ____________________________________ ;    //LINE-2
    return _______________ ;                  //LINE-3
}

int main() {
    int n;
    cin >> n;
    cout << findNthPrime(n) << endl;
    return 0;
}
```

Public 1

`Input: 150`

```
Output: 863
```

Public 2

```
Input: 200
Output: 1223
```

Private

```
Input: 1000
Output: 7919
```

Answer:
```
LINE-1:  long long operator()()
LINE-2:  auto a = async(NthPrime(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 `NthPrime()` can be made as:
`auto a = async(NthPrime(n))`
At `LINE-3` can use the statement:
`a.get()`
to receive the result of functor `NthPrime()`.