Programming in Modern C++: Assignment Week 10

Total Marks : 25

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

Question 1

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

```
#include<iostream>
int main( ){
    char n = 'A';
    char& rn = n;
    __________ t = rn;    //LINE-1

    ++t;
    std::cout << n << " " << rn << " " << t << std::endl;
    return 0;
}
```

Identify the appropriate option/s to fill in the blank at `LINE-1` such that output becomes `B B B`

a) `auto`

b) `auto&`

c) `decltype(rn)`

d) `decltype(n)`

Answer: b), c)
Explanation:
Since `auto` never deduces adornments like cv-qualifer or reference, in option a) the inferred type of `t` is `char`. Thus, output will be `A A B`.
In option b) the inferred type of `t` is `char&`. Thus, output will be `B B B`.
In option c) the inferred type of `t` is the type of `rn` that is `char&`. Thus, output will be `B B B`.
In option d) the inferred type of `t` is the type of `n` that is `char`. Thus, output will be `A A B`.

Question 2

Consider the code segment (in C++14) given below. *[MSQ, Marks 2]*

```
#include<iostream>
struct Oper1{
    int i;
    Oper1(int _i) : i(_i){}
    int& operator()() { std::cout << "1 "; return i ; }
};

struct Oper2{
    int i;
    Oper2(int _i) : i(_i){}
    int operator()() { std::cout << "2 "; return i ; }
};


template < typename U >
__________________ {    //LINE-1
    return op() ;
}

int main(){
    Oper1 o1{10};
    Oper2 o2{10};
    foobar(o1) = 20;
    foobar(o2) ;
    return 0;
}
```

Identify the appropriate option/s to fill in the blank at `LINE-1` such that output becomes `1 2`.

a) `auto foobar( U& op ) -> decltype(op())`

b) `auto foobar( U& op )`

c) `auto& foobar( U& op )`

d) `decltype(auto) foobar( U& op )`

Answer: a), d)
Explanation:
The call `foobar(o1) = 20;` evaluates to lvalue of type `int&`.
The call `foobar(o2);` evaluates to prvalue of type `int`.
Since plain `auto` never deduces to a reference, option b) fails for prvalue.
Since plain `auto&` always deduces to a reference, option b) fails for prvalue.
Option a) and d) works for `lvalue` as well as `prvalue`. Thus these two are correct options.

Question 3

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

```
class CustList{
    public:
        CustList(std::initializer_list<double> dlist) { cout << "ctor-1" << " "; }
        CustList(std::initializer_list<int> dlist) { cout << "ctor-2" << " "; }
        CustList(double d1, double d2, double d3) { cout << "ctor-3" << " "; }
};
```

Indemnify the appropriate option that present all correct output/error for the following instantiation of `CustList` class:

1. `CustList c{3.1, 4.5, 6.5};`
2. `CustList c(3.1, 4.5, 6.5);`
3. `CustList c{3.1f, 4.5f, 6.5};`
4. `CustList c{3, 4.5, 6};`

a) (a) `ctor-1`
   (b) `ctor-3`
   (c) `ctor-1`
   (d) `ctor-3`

b) (a) `ctor-1`
   (b) `ctor-3`
   (c) `ctor-1`
   (d) `compiler error: call of overloaded 'CustList()' is ambiguous for CustList c{3, 4.5, 6};`

c) (a) `ctor-1`
   (b) `ctor-1`
   (c) `ctor-1`
   (d) `compiler error: call of overloaded 'CustList()' is ambiguous for CustList c{3, 4.5, 6};`

d) (a) `ctor-3`
   (b) `ctor-1`
   (c) `compiler error: call of overloaded 'CustList()' is ambiguous for CustList c{3.1f, 4.5f, 6.5};`
   (d) `compiler error: call of overloaded 'CustList()' is ambiguous for CustList c{3, 4.5, 6};`

Answer: b)
Explanation:
The statement `CustList c{3.1, 4.5, 6.5};` invokes constructor:

```
CustList(std::initializer_list<double> dlist);
```

The statement `CustList c(3.1, 4.5, 6.5);` invokes constructor:

```
CustList(std::initializer_list<int> dlist);
```

Since conversion `float => double` is better than `double => int` the statement `CustList c{3.1f, 4.5f, 6.5};` invokes constructor:

```
CustList(double d1, double d2, double d3)
```

Since the `int => double` and `double => int` have the same rank the call `CustList c{3, 4.5, 6};` is ambiguous.

Question 4

Consider the following code segment. *[MSQ, Marks 2]*

```
#include <iostream>

class ComplexNum{
    public:
        constexpr ComplexNum(int _r = 0, int _i = 0) : r(_r), i(_i){ }
    private:
        int r, i;
};
int randGen(){
    return 10;
}
constexpr int numGen(int i, int j){
    return i + j;
}
int main(){
    constexpr ComplexNum c1(10, 20);    //LINE-1
    constexpr int i = 10, j = 20;
    constexpr ComplexNum c2(i, j);      //LINE-2
    constexpr ComplexNum c3(randGen(), randGen());          //LINE-3
    constexpr ComplexNum c4(numGen(i, j), numGen(i, j));    //LINE-4
    return 0;
}
```

Which of the following line/s generate/s compiler error?

a) `LINE-1`

b) `LINE-2`

c) `LINE-3`

d) `LINE-4`

Answer: c)
Explanation:
At `LINE-1`, the constructor parameters are constants from the compile time context. Thus, the object can be `constexpr`.
At `LINE-2`, the constructor parameters are `constexpr`. Thus, the object can be of `constexpr`.
At `LINE-3`, the constructor parameters are the return value of the `randGen()` function. However, the return type of `randGen()` is not `constexpr`, therefore the object cannot be `constexpr`.
At `LINE-4`, the constructor parameters are the return value of the `numGen(int, int)` function. However, the return type of `numGen(int, int)` is `constexpr`, therefore the object can be `constexpr`.
Intentionally made MSQ.

Question 5

Consider the following code segment. *[MSQ, Marks 2]*

```
#include <iostream>

void update(char* str){ /*some code*/ }

template<typename F, typename P>
void caller(F func, P s){
    func(s);
}

int main(){
    char s[2] = "0";
    char *p = &s[1];
    caller(update, p);          //LINE-1
    caller(update, 0);          //LINE-2
    caller(update, NULL);       //LINE-3
    caller(update, nullptr);    //LINE-4
    return 0;
}
```

Which of the following lines generate/s compiler error?

a) `LINE-1`

b) `LINE-2`

c) `LINE-3`

d) `LINE-4`

Answer: b), c)
Explanation:
For the call in `LINE-1`, the template type parameter `P` is deduced to `char*`. Thus, it does not generate any compiler error.
For the call in `LINE-2`, the template type parameter `P` is deduced to `int`. Thus, it generates a compiler error.
For the call in `LINE-3`, the template type parameter `P` is deduced to `long int`. Thus, it generates a compiler error.
For the call in `LINE-2`, the template type parameter `P` is deduced to `std::nullptr_t` and the call `update(std::nullptr_t)` is syntactically correct.

Question 6

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

```
#include<iostream>
#include<iomanip>

_____________________________________ {    //LINE-1
   return 1024 * mem;
}

_____________________________________ {    //LINE-2
   return mem;
}
int main() {
   long double size = 10.0_KB + 2.0_B;
   std::cout << "size (in bytes): " << size;
   return 0;
}
```

Identify the appropriate option to fill in the blanks at `LINE-1` and `LINE-2` such that the output becomes `size (in bytes):  10242`.

a)
```
LINE-1:  long double operator"" KB(long double mem)
LINE-2:  long double operator"" B(long double mem)
```

b)
```
LINE-1:  long double operator"" _KB(long double mem)
LINE-2:  long double operator"" _B(long double mem)
```

c)
```
LINE-1:  long int operator"" _KB(long int mem)
LINE-2:  long int operator"" _B(long int mem)
```

d)
```
LINE-1:  unsigned long long operator _KB(unsigned long long mem)
LINE-2:  unsigned long long operator _B(unsigned long long mem)
```

Answer: b)
Explanation:
Since the user-defined literals in the program are double type option b) is correct.

Question 7

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

```
#include<iostream>
#include<vector>
#include<cmath>

void process(int& v) {
    if(v < 0)
        throw v;
    ++v;
}

void func(std::vector<int>& iVec) noexcept(noexcept(process(iVec[0]))){
    for(int& v : iVec)
        process(v);
}

int main() {
    std::vector<int> iVec{1, 2, -1, 2};
    try{
        func(iVec);
    }catch(int i){
    }
    for(int v : iVec)
        std::cout << v << " ";
    return 0;
}
```

Identify the correct option about the program above.

a) It generates output as `2 3 0 3`

b) It generates output as `2 3 -1 2`

c) It generates output as `2 3 -1 3`

d) The program gets terminated since a function that is declared `noexcept` throws an exception

Answer: b)
Explanation:
If we consider the function header:
`void func(std::vector<int>& iVec) noexcept(noexcept(process(iVec[0])))`
Since the function `process(iVec[0])` is not declared as `noexcept`, `noexcept(process(iVec[0]))` is false. Thus the function header is treated as:
`void func(std::vector<int>& iVec) noexcept(false)`
Therefore, since for function`func` `noexcept` is false, it can throw exception. The values `1 2` will become `2 3`. However, for `-1`, it throws exception and control comes out of `try` block (in `main()`).

Question 8

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

```
#include<iostream>
int i = 10;

void test(int&& rv){ }

int getVal(){
    return i;
}

int& getRef(){
    return i;
}

int main() {
    test(i);           //LINE-1
    test(i + 10);      //LINE-2
    test(getVal());    //LINE-3
    test(getRef());    //LINE-4
    return 0;
}
```

Identify the line/s generate/s compiler error.

a) `LINE-1`

b) `LINE-2`

c) `LINE-3`

d) `LINE-4`

Answer: a), d)
Explanation:
The function `test(int&&)` accepts an integer which is a non-constant rvalue.
At `LINE-1`, we pass a lvalue `i`, so it is incorrect.
At `LINE-2`, `i + 10` is a temporary value which is a rvalue. Therefore, the call is correct.
At `LINE-3`, the function `getVal()` returns a rvalue. Therefore, the call is correct.
At `LINE-4`, the function `getRef()` return a reference which is a lvalue. Therefore, the call is incorrect.

Question 9

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

```
#include<iostream>

class Resource {
    public:
        Resource() { std::cout << "#1" << " "; }
        Resource(const Resource&) { std::cout << "#2" << " "; }
        Resource(Resource&&) noexcept { std::cout << "#3" << " "; }
        Resource& operator=(const Resource&) { std::cout << "#4" << " ";
                                return *this; }
        Resource& operator=(Resource&&) noexcept { std::cout << "#5" << " ";
                                return *this; }
};

Resource createResource(){
    Resource r;
    return r;
}

int main() {
    Resource r1;
    r1 = createResource();
    Resource r2 = r1;
    Resource r3 = std::move(r2);
    return 0;
}
```

What will be the output?

a) `#1 #5 #4 #3`

b) `#1 #1 #5 #4 #3`

c) `#1 #3 #2 #3`

d) `#1 #1 #5 #2 #3`

Answer: d)
Explanation:
The statement `Resource r1;` calls default constructor and prints `#1`.
In the statement `r1 = createResource();`, the function `createResource()` calls default constructor and prints `#1`. Then, it calls move assignment constructor and prints `#5`.
The statement `Resource r2 = r1;` calls copy constructor and prints `#2`.
The statement `Resource r3 = std::move(r2);` move constructor and prints `#3`.

Programming Questions

Question 1

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

- Fill in the blanks at `LINE-1` and `LINE-2` with appropriate namespace declaration.
- Fill in the blanks at `LINE-3` with an appropriate statement, such that all symbols defined in the scope `namespace Ver1` becomes available in the `main()`.

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

```
#include <iostream>
#include <list>
namespace Ver1{
    ____________________ {    //LINE-1
        int addAll(std::list<int> i_list){
            int sum = 0;
            for(auto i : i_list){
                sum += i;
            }
            return sum;
        }
    }
    ____________________ {    //LINE-2
        template<typename T>
        T addAll(std::list<T> t_list){
            T sum = 0;
            for(auto i : t_list){
                sum += i;
            }
            return sum;
        }
    }
}

______________________ ;    //LINE-3
int main(){
    int n;
    std::cin >> n;
    std::list<int> ilist;
    std::list<double> dlist;
    for(int i = 0; i < n; i++){
        int x;
        std::cin >> x;
        ilist.push_back(x);
    }
    for(int i = 0; i < n; i++){
        double x;
        std::cin >> x;
        dlist.push_back(x);
    }
    std::cout << Ver1_1::addAll(ilist) << " ";
```

```
    std::cout << addAll(ilist) << " ";
    std::cout << addAll(dlist);
    return 0;
}
```

Public 1

```
Input:
3
2 4 6
2.1 3.4 5.2
Output:
12 12 10.7
```

Public 2

```
Input:
5
3 9 4 8 3
4.3 2.5 6.5 1.2 7.4
Output:
27 27 21.9
```

Private

```
Input:
4
7 3 4 5
4.3 2.4 5.6 7.3
Output:
19 19 19.6
```

Answer:

```
LINE-1:  namespace Ver1_1
LINE-2:  inline namespace Ver1_2
LINE-3:  using namespace Ver1
```

Explanation:

Since the integer version function `addAll` is explicitly called from `namespace Ver1_1`, the blank at `LINE-1` must be filled in as:

`namespace Ver1_1`

Since the template version of function `addAll` is called by default, the blank at `LINE-2` must be filled in as:

`inline namespace Ver1_2`

Please note that namespace name can be different in this case.

Since we access the namespaces from `Ver1` directly in `main()`, the blank at `LINE-3` must be filled in as:

`using namespace Ver1`

Question 2

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

- Fill in the blank at `LINE-1` with an appropriate template definition.
- Fill in the blank at `LINE-2` with an appropriate header for function `divide`.

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

```
#include <iostream>
int getNumber(char c){
    return int(c);
}
int getNumber(double d){
    return int(d);
}
double getNumber(int i){
    return double(i);
}

------------------------------    //LINE-1
------------------------------ {   //LINE-2
    return getNumber(n1) / getNumber(n2);
}
int main(){
    int a;
    double b;
    char c;
    std::cin >> a >> b >> c;
    std::cout << divide(c, a) << " ";
    std::cout << divide(c, b);
    return 0;
}
```

Public 1

```
Input:
10 2.0 A
Output:
6.5 32
```

Public 2

```
Input:
5 5.0 B
Output:
13.2 13
```

Private

```
Input:
10 10.5 C
Output:
6.7 6
```

Answer:
In C++11

```
LINE-1: template<typename T, typename U>
LINE-2: auto divide(T n1, U n2) -> decltype(getNumber(n1) / getNumber(n2))
```

or in C++14

```
LINE-2: decltype(auto) divide(T n1, U n2)
```

Explanation:
Since we can pass two different types of parameters in `divide` function, we can write at `LINE`:
`template<typename T, typename U>`
The header for function `divide` in C++11 should be:
`auto divide(T n1, U n2) -> decltype(getNumber(n1) / getNumber(n2))`
And in C++14:
`decltype(auto) divide(T n1, U n2)`

Question 3

Consider the following program that implements copy constructor, copy assignment, move constructor and move assignment.

- Fill the missing code segments at `code-segment-1` to implement the move constructor.
- Fill the missing code segments at `code-segment-2` to implement the move assignment.

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

```
#include <iostream>
class point {
    public:
        point(int x = 0, int y = 0) : _px(new int(x)), _py(new int(x)) { }
        point(const point& p) : _px(new int(*(p._px) * 2)),
                                _py(new int(*(p._py) * 2)) { }
        point& operator=(const point& p) {
        if (this != &p) {
            delete _px;
            delete _py;
            _px = new int(*(p._px) * 3);
            _py = new int(*(p._py) * 3);
        }
        return *this;
    }
    ~point() { delete _px; delete _py; }
    point(point&& p) noexcept : _px(p._px), _py(p._py) {
        //code-segment-1
    }
    point& operator=(point&& p) noexcept{
        //code-segment-2
    }
    friend std::ostream& operator<<(std::ostream& os, const point& p) {
        std::cout << "(" << *(p._px) << ", " << *(p._py) << ")";
        return os;
    }
    friend std::istream& operator>>(std::istream& os, point& p) {
        std::cin >> *(p._px) >> *(p._py);
        return os;
    }
    private:
        int *_px = nullptr, *_py = nullptr;
};

int main(){
    point p1;
    std::cin >> p1;
    point p2 = p1;
    point p3;
    p3 = p1;
    std::cout << p1 << ", " << p2 << ", " << p3 << std::endl;

    point p4 = std::move(p1);
```

```
    std::cout << p4 << ", ";
    point p5;
    p5 = std::move(p4);
    std::cout << p5;
    return 0;
}
```

Public 1

```
Input: 10 10
Output:
(10, 10), (20, 20), (30, 30)
(40, 40), (200, 200)
```

Public 2

```
Input: 1 2
Output:
(1, 2), (2, 4), (3, 6)
(4, 8), (20, 40)
```

Private

```
Input: 100 100
Output:
(100, 100), (200, 200), (300, 300)
(400, 400), (2000, 2000)
```

Answer:

```
code-segment-1:

*_px *= 4;
*_py *= 4;
p._px = nullptr;
p._py = nullptr;

code-segment-2:

if (this != &p) {
    delete _px;
    delete _py;
    _px = p._px;
    _py = p._py;
    *_px *= 5;
    *_py *= 5;
    p._px = nullptr;
    p._py = nullptr;
}
return *this;
```

Explanation:
The implementation of move constructor is as follows:

```
point(point&& p) noexcept : _px(p._px), _py(p._py) {
    *_px *= 4;
    *_py *= 4;
    p._px = nullptr;
    p._py = nullptr;
}
```

The implementation of move assignment is as follows:

```
point& operator=(point&& p) noexcept{
    if (this != &p) {
        delete _px;
        delete _py;
        _px = p._px;
        _py = p._py;
        *_px *= 5;
        *_py *= 5;
        p._px = nullptr;
        p._py = nullptr;
    }
    return *this;
}
```