Programming in Modern C++: Assignment Week 10

Total Marks : 20

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

September 22, 2023

Question 1

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

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

int main( ){
    std::list<int> li { 10, 20, 30, 40, 50 };
    int i = 0;
    for(________________________________________)    // LINE-1
        it *= 2;

    for(auto it = li.begin(); it != li.end(); it++)
        std::cout << *it << " ";
    return 0;
}
```

Identify the appropriate option(s) to fill in the blank at `LINE-1` such that the output of the program is: `20 40 60 80 100`

a) `auto it :  li`

b) `auto& it :  li`

c) `decltype(i) it :  li`

d) `decltype((i)) it :  li`

Answer: b), d)
Explanation:
In option a), the inferred type of `it` is `int`. Thus, the changes made in `it` are not reflected in the list `li`. So, the O/P is `10 20 30 40 50`
In option b), the inferred type of `it` is `int&`. Thus, the changes made in `i` are reflected in the list `li`.
In option c), the inferred type of `it` is the output of `decltype(i)` i.e. `int`. Thus, the changes made in `i` are not reflected in the list `li`.
In option d), the inferred type of `it` is the output of `decltype((i))` i.e. `int&`. Thus, the changes made in `it` are reflected in the list `li`.

Question 2

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

```
#include <iostream>

int main( ){
    int n = 10;

    int& i1 = n;
    const int& i2 = 10;

    auto x1 = i1;
    auto x2 = i2;
    decltype(i1) x3 = i1;
    decltype(i2) x4 = i1;

    ++x1;    //LINE-1
    ++x2;    //LINE-2
    ++x3;    //LINE-3
    ++x4;    //LINE-4

    return 0;
}
```

Which of the following line/lines generate/generates compiler error?

a) `LINE-1`

b) `LINE-2`

c) `LINE-3`

d) `LINE-4`

Answer: d)
Explanation:
Since `auto` never deduces adornments like cv-qualifer or reference (however, no error or exception is generated), the inferred type of `x1` and `x2` is `int`. For `x3`, the inferred type is `int&`, whereas for `x4`, the inferred type is `const int&`. Therefore, d) is the correct option.
Intentionally kept as MSQ

Question 3

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

```
#include<iostream>

int x = 10;
struct operation1 {
    operation1(int val) : val_(val){}
    int& operator()() { std::cout << val_ << " "; return x; }
    int val_;
};

struct operation2 {
    operation2(int val) : val_(val){}
    int operator()() { std::cout << val_ << " "; return x; }
    int val_;
};


template <typename T>
_________________________________ {    //LINE-1
    return op() ;
}

int main(){
    operation1 o1{1};
    operation2 o2{2};
    wrapper(o1) = 10;
    int i = wrapper(o2);
    return 0;
}
```

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

a) `auto wrapper( T& op ) -> decltype(op())`

b) `auto wrapper( T& op )`

c) `auto& wrapper( T& op )`

d) `decltype(auto) wrapper ( T& op )`

Answer: a), d)
Explanation:
The call `wrapper(o1) = 10;` evaluates to lvalue of type `int&`.
The call `wrapper(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 4

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

```
#include<iostream>

constexpr int f2(const int i){
    return i + 10;
}

void f1(const int i){
    constexpr int n = 20;
    constexpr int c1 = n + 30;      //LINE-1
    constexpr int c2 = n + c1;      //LINE-2
    constexpr int c3 = n + i;       //LINE-3
    constexpr int c4 = n + f2(i);   //LINE-4
}

int main(){
    f1(10);
    return 0;
}
```

Identify the line/lines generate/generates compiler error.

a) `LINE-1`

b) `LINE-2`

c) `LINE-3`

d) `LINE-4`

Answer: c), d)
Expanation:
`constexpr` needs compile-time constant.
At `LINE-1`, c1 = n + 30; where `n` is a `constexpr` and `30` is a literal. Therefore, `c1` is a `constexpr`.
At `LINE-2`, c2 = n + c1 where `n` and `c1` both are `constexpr`. Therefore, `c2` is a `constexpr`.
At `LINE-3`, c3 = n + i; where `n` is a `constexpr`; however `i` is not a compile-time constant. Therefore, `c2` cannot be `constexpr`.
At `LINE-4`, c4 = n + f2(i); where `n` is a `constexpr`; however the call `f2(i)` fails since `i` is not a compile-time constant.

Question 5

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

```
#include <iostream>
#include <vector>
#include <initializer_list>

template<typename T>
class Numbers{
    public:
        Numbers() { std::cout << "cont-1" << " "; }
        Numbers(int n) { std::cout << "cont-2" << " "; }
        Numbers(std::initializer_list<int> elems) { std::cout << "cont-3" << " "; }
        Numbers(int n, std::initializer_list<int> elms) { std::cout << "cont-4" << " ";
    }
};
int main(){
    Numbers<int> n1(10);
    Numbers<int> n2({10, 20, 30});
    Numbers<int> n3{10, 20, 30};
    Numbers<int> n4 = {10, 20, 30};
    Numbers<int> n5(10, {10, 20, 30});
    return 0;
}
```

What will be the output?

a) `cont-2 cont-3 cont-3 cont-3 cont-4`

b) `cont-2 cont-3 cont-3 cont-1 cont-4`

c) `cont-2 cont-3 cont-2 cont-1 cont-3`

d) `cont-2 cont-3 cont-3 cont-3 cont-3`

Answer: a)
Explanation:
`Numbers<int> n1(10);` invokes parameterized constructor `Numbers(int n) { ... }`.
`Numbers<int> n2({10, 20, 30});`, `Numbers<int> n3{10, 20, 30};` and `Numbers<int> n4 = {10, 20, 30};` invoke the initializer list constructor `Numbers(initializer_list<int> elms){ ... }`.
`Numbers<int> n5(10, {10, 20, 30});` invokes the mixed constructor `Numbers(int n, initializer_list<int> elms){ ... }`.

Question 6

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

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

long double operator""_FT(long double n) {
    return n * 12;
}

long double operator"" _IN(long double n) {
    return n;
}

int main() {
    long len = _______________________________;    //LINE-1
    std::cout << len << "IN";
    return 0;
}
```

Choose the appropriate option to fill in the blank at `LINE-1`, such that the output becomes `80IN`.

a) `6.0FT + 8.0IN`

b) `6.0_FT + 8.0_IN`

c) `(FT)6.0 + (IN)8.0`

d) `6_FT + 8_IN`

Answer: b)
Explanation: For user-defined numeric literal operators, the correct way to invoke them is to write them as `6.0_FT + 8.0_IN`.
All other options are compilation error. Even option d) is wrong as numeric literal operators require exact type matching
Intentionally kept as MSQ

Question 7

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

```
#include <iostream>

_______________________________ {           // LINE-1
    double divide(double n){
        return n / 10;
    }
}

_______________________________ {           // LINE-2
    template<typename T>
    T divide(T n){
        return n / 100;
    }
}

int main(){
   std::cout << ver1_0::divide(100.0) << " ";
   std::cout << ver1_1::divide(100) << " ";
   std::cout << divide(100.0);
   return 0;
}
```

Choose the appropriate option to fill in the blanks at `LINE-1` and `LINE-2` so that the output becomes
`10 1 1`

a) `LINE-1:  namespace ver1_0`
   `LINE-2:  namespace ver1_1`

b) `LINE-1:  namespace ver1_0`
   `LINE-2:  inline namespace ver1_1`

c) `LINE-1:  inline namespace ver1_0`
   `LINE-2:  namespace ver1_1`

d) `LINE-1:  inline namespace ver1_0`
   `LINE-2:  inline namespace ver1_1`

Answer: b)
Explanation:
As per the output of `ver1_0::divide(10, 4)` and `ver1_1::divide(10, 4)`, the `ver1_0` and `ver1_0` must have basic namespace definition. However, since `divide(10, 4)` invoke the function `divide` from `ver1_1`, `ver1_1` must be the default namespace. Thus, at `LINE-1` and `LINE-2`, we must have:

```
namespace ver1_0
inline namespace ver1_1
```

Question 8

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

```
#include <iostream>

void show(int* ip){ /* code */ }

template<typename Func, typename Param>
void call(Func fn, Param p){
    fn(p);
}

int main(){
    int i = 10;
    call(show, &i);          //LINE-1
    call(show, i);           //LINE-2
    call(show, NULL);       //LINE-3
    call(show, nullptr);    //LINE-4
    return 0;
}
```

Choose the call/s to `call` function that will result in compiler error/s.

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 `Param` is deduced to `int*`. Thus, it does not generate any compiler error.
For the call in `LINE-2`, the template type parameter `Param` is deduced to `int`. Thus, it generates a compiler error.
For the call in `LINE-3`, the template type parameter `Param` is deduced to `long int` (which is the datatype of `NULL`). Thus, it generates a compiler error.
For the call in `LINE-4`, the template type parameter `Param` is deduced to `std::nullptr_t` and the call `show(std::nullptr_t)` is syntactically correct.

Question 9

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

```
#include<iostream>
#include<utility>

class number{
    public:
        number(const int& i = 0) : i_(i) { }
        number(const number& ob) {
            std::cout << "number-cp-ctor" << " ";
        }
        number(number&& ob) noexcept {
            std::cout << "number-mv-ctor" << " ";
        }
    private:
        int i_;
};

class sp_number : public number{
    public:
        sp_number(const int& i) : number(i) { }
        sp_number(const sp_number& ob) : number(ob) {
            std::cout << "spnumber-cp-ctor" << " ";
        }
        sp_number(sp_number&& ob) noexcept : number(ob) {
            std::cout << "spnumber-mv-ctor" << " ";
        }
};

int main(){
    sp_number obj1(100);
    sp_number obj2(obj1);
    sp_number obj3(std::move(obj1));
    return 0;
}
```

What will be the output?

a) `number-cp-ctor spnumber-mv-ctor number-cp-ctor spnumber-mv-ctor`

b) `number-cp-ctor spnumber-cp-ctor number-mv-ctor spnumber-cp-ctor`

c) `number-cp-ctor spnumber-cp-ctor number-mv-ctor spnumber-mv-ctor`

d) `number-cp-ctor spnumber-cp-ctor number-cp-ctor spnumber-mv-ctor`

Answer: d)
Explanation:
Since the constructors are invoked in a top-down order in C++ class hierarchy, the construction for the given program takes place as follows:
The statement `sp_number obj2(obj1);` calls the copy constructor of `sp_number`, which forward the call to the copy constructor of `number`. Thus, it prints
`number-cp-ctor spnumber-cp-ctor`

The statement `sp_number obj3(std::move(obj1));` calls the move constructor of `sp_number`, which forward the call to the copy constructor of `number`. Thus, it prints

```
number-cp-ctor spnumber-mv-ctor
```

Programming Questions

Question 1

Consider the following program in C++11/14 to convert between feet and inch. Fill in the blanks as per the instructions given below:

- at `LINE-1` with appropriate header to function `convert`,
- at `LINE-2` with appropriate return statement `convert`,

such that it will satisfy the given test cases. *Marks: 3*

```
#include <iostream>

class feet;

class inch{
    public:
        inch(double i) : i_(i){}
        feet getValue();
        void show(){ std::cout << i_ << " "; }
    private:
        double i_;
};

class feet{
    public:
        feet(double f) : f_(f){}
        inch getValue();
        void show(){ std::cout << f_ << " "; }
    private:
        double f_;
};

feet inch::getValue(){
    feet t(i_ / 12.0);
    return t;
}

inch feet::getValue(){
    inch t(f_ * 12.0);
    return t;
}

template <typename T>
_________________________________________________  {     // LINE-1
    __________________________________;                  // LINE-2
}

int main(){
    double a, b;
    std::cin >> a >> b;
    feet f(a);
    inch i(b);
    inch i1 = convert(f);
    feet f1 = convert(i);
    i1.show();
    f1.show();
    return 0;
}
```

Public 1

```
Input: 12 12
Output: 144 1
```

Public 2

```
Input: 90 90
Output: 1080 7.5
```

Private

```
Input: 12 120
Output: 144 10
```

Answer:

```
LINE-1:  auto convert(T n) -> decltype(n.getValue()) //in C++11
LINE-1:  decltype(auto) convert(T n) //in C++14
LINE-2:  return n.getValue()
```

Explanation:

The function `convert` must have a trailing return type. Thus, the header of `convert` function should be:

```
auto convert(T n) -> decltype(getValue(n)) //in C++11
decltype(auto) convert(T n) //in C++14
```

The body of the function should be:

```
return getValue(n);
```

Note that return type of `convert()` is different from its parameter type `T` and depends on the return type of `getValue()`. So it is difficult to write this template in C++03

Question 2

Consider the following program in C++11/14. Fill in the blanks as per the instructions given below:

- at LINE-1 with appropriate header and initialization list for the copy constructor,
- at LINE-2 with appropriate header for copy assignment operator overload,
- at LINE-3 with appropriate header and initialization list for the move constructor,
- at LINE-4 with appropriate header for move assignment operator overload,

such that it will satisfy the given test cases. *Marks: 3*

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

class number {
    public:
        number(){}
        number(int i) : ip_(new int(i)) { }
        ______________________________ { }    // LINE-1: copy constructor
        ______________________________ {      // LINE-2: copy assignment
            if (this != &n) {
                delete ip_;
                ip_ = new int(*(n.ip_) * 10);
            }
            return *this;
        }
        ~number() { delete ip_; }
        ____________________________ { n.ip_ = nullptr; }  // LINE-3: move constructor
        _______________________________ {        // LINE-4: move assignment
            if (this != &d) {
                ip_ = d.ip_;
                d.ip_ = nullptr;
            }
            return *this;
        }
        void show(){
            if(ip_ == nullptr)
                std::cout << "moved : ";
            else
                std::cout << *ip_ << " : ";
        }
        private:
            int* ip_ {nullptr};
};

int main(){
    int a;
    std::cin >> a;
    number n1(a);
    number n2 = n1;
    number n3;
    n3 = n1;
    n1.show();
    n2.show();
    n3.show();

    number n4 = std::move(n1);
    number n5;
    n5 = std::move(n1);
    n1.show();
    n4.show();
    n5.show();
    return 0;
}
```

Public 1

```
Input: 5
Output: 5 : 50 : 50 : moved : 5 : moved :
```

Public 2

```
Input: -10
Output: -10 : -100 : -100 : moved : -10 : moved :
```

Private

```
Input: 1
Output: 1 : 10 : 10 : moved : 1 : moved :
```

Answer:

```
LINE-1:  number(const number& n) :  ip_(new int(*(n.ip_) * 10))
LINE-2:  number& operator=(const number& n)
LINE-3:  number(number&& n) :  ip_(n.ip_)
LINE-4:  number& operator=(number&& d)
```

Explanation:
As per the output specified, the header and initialization list for copy constructor at `LINE-1` is:
`number(const number& n) :  ip_(new int(*(n.ip_) * 10))`,
the header for copy assignment operator for copy assignment is:
`number& operator=(const number& n)`,
the header and initialization list for move constructor at `LINE-3` is:
`number(number&& n) :  ip_(n.ip_)`,
the header for move assignment at `LINE-4` is:
`number& operator=(number&& d)`.