Programming in Modern C++: Assignment Week 7

Total Marks : 20

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

September 1, 2023

Question 1

Consider the following program. *[MCQ, Marks 2]*

```
#include<iostream>
using namespace std;
class A{
    public:
        virtual void f() {}
        void g() {}
};
class B : public A{
    public:
        void g() {}
        virtual void h() {}
        virtual void i();
};
class C : public B{
    public:
        void f() {}
        virtual void h() {}
};
```

What will be the virtual function table (VFT) for the `class C`?

a)
```
C::f(C* const)
C::h(C* const)
B::i(B* const)
```

b)
```
A::f(A* const)
B::g(B* const)
C::h(C* const)
B::i(B* const)
```

c)
```
A::f(A* const)
B::h(B* const)
C::i(C* const)
```

d) `A::f(A* const)`
`B::g(B* const)`
`C::h(C* const)`
`C::i(C* const)`

Answer: a)
Explanation:
All three functions except `g` are virtual in the `class C`. So, there will be three entries in the virtual function table.
Now, function `f()` is overridden in `class C`. So, the entry for function `f()` in the virtual function table of `class C` will be `C::f(C* const)`.
The function `h()` is declared as virtual in `class C`. So, the entry for function `h()` in VFT of class C will be `C::h(C* const)`.
The function `i()` is declared as virtual in `class B`. But not overridden in `C`. So, the entry for function `i()` in VFT of `class C` will be `B::i(B* const)`.

Question 2

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

```
#include <iostream>
using namespace std;
int main() {
    char c = 'Z';
    int i = 50;
    char *cp = &c;
    int *pd;
    c = static_cast<char>(i);      // LINE-1
    i = static_cast<double>(c);    // LINE-2
    pd = static_cast<double*>(cp); // LINE-3
    c = static_cast<char>(&c);     // LINE-4
    return 0;
}
```

Which line/s will give you an error?

a) `LINE-1`

b) `LINE-2`

c) `LINE-3`

d) `LINE-4`

Answer: c), d)
Explanation:
`static_cast` cannot cast between two different pointer types. In `LINE-3`, `int*` is assigned to `char*`. Hence, it is an error.
Using `static_cast`, it is not possible to change a pointer type to a value type. In `LINE-4`, `char*` is assigned to `char` which is not possible using static cast.

Question 3

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

```
#include <iostream>
using namespace std;
double incr(double* ptr){
    return (*ptr)++;
}
int main() {
    double val = 3.14;
    const double *ptr = &val;
    val = incr(_________________________); //LINE-1
    cout << val;
    return 0;
}
```

Fill in the blank at `LINE-1` such that the program will print: `3.14`.

a) `const_cast<double*>(ptr)`

b) `static_cast<double*>(ptr)`

c) `dynamic_cast<double*>(ptr)`

d) `reinterpret_cast<double*>(ptr)`

Answer: a)
Explanation:
The function `incr()` modifies the value of `*ptr`, but the previous value is returned as return-by-value. But, in `main()` function, `ptr` is declared as `const double *ptr;`. Hence, the constant-ness of `*ptr` has to be removed, which can be done using `const_cast`. So, a) is the correct option.

Question 4

Consider the code segment given below. *[MCQ, Marks 2]*

```
class M { public: virtual void f() { } };
class N : public M { };
class O : public M { public: void g() {} };
class P : public N, public O{ public: void g(){ }};
```

How many virtual tables will be created?

a) `1`

b) `2`

c) `3`

d) `4`

Answer: d)
Explanation:
The presence of a virtual function (either explicitly declared or inherited from a base class) makes the class polymorphic. For such classes, we need a class-specific virtual function table (VFT). All four classes, thus, will set up virtual function tables.

Question 5

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

```
class C1 { int a=1; };
class C2 { int b=2; };
C1* s1 = new C1;
C2* s2 = new C2;
```

Which of the following type-casting is permissible?

a) `s2 = static_cast<C2*>(s1);`

b) `s2 = dynamic_cast<C2*>(s1);`

c) `s2 = const_cast<C2*>(s1);`

d) `s2 = reinterpret_cast<C2*>(s1);`

Answer: d)
Explanation:
On each option, there is an attempt to cast from `C1*` to `C2*`, and these two classes are unrelated. As we know, only reinterpret cast can be used to convert a pointer to an object of one type to a pointer to another object of an unrelated type. Hence, only option d) is correct.

Question 6

Consider the code segment given below. *[MCQ, Marks 2]*

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Base { public: virtual ~Base(){}};
class Derived: public Base {};
int main() {
    Base b; Derived d;
    Derived *dp = &d;
    Base *bp = dp;
    Derived *dpp = (Derived*)dp;
    cout << (typeid(dp).name() == typeid(bp).name());
    cout << (typeid(*dp).name() == typeid(*bp).name());
    cout << (typeid(bp).name() == typeid(dpp).name());
    cout << (typeid(*bp).name() == typeid(*dpp).name());
    cout << (typeid(*dp).name() == typeid(*dpp).name());
    return 0;
}
```

What will be the output?

a) `00011`

b) `01001`

c) `01011`

d) `00110`

Answer: c)
Explanation:
Type of `dp` is `Derived*` and type of `bp` is `Base*`. Thus, output is 0.
`*dp` and `*bp` point to the same object `d`, and it is a dynamic binding situation. Thus, both are of type Derived, and output is 1.
Type of `bp` is `Base*` and type of `dpp` is `Derived*`. Thus, output is 0.
`*bp` and `*dpp` point to the same object `d`, and it is a dynamic binding situation. Thus, both are of type Derived, and output is 1.
`*dp` and `*dpp` point to the same object `d`, and it is a dynamic binding situation. Thus, both are of type Derived, and output is 1.

Question 7

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

```
#include <iostream>
using namespace std;
class book{
    string _title;
    string _author;
    string _publisher;
    public:
        book(string title, string author, string publisher)
            : _title(title), _author(author), _publisher(publisher){}
        void changePublisher(string new_publisher) const{
            (_____________)->_publisher = new_publisher; //LINE-1
        }
        void book_details() const{
            cout << _title << ":" << _author << ":" << _publisher;
        }
};
int main(){
    const book b("Modern C++", "Partha Pratim Das", "IIT KGP");
    b.changePublisher("IITM");
    b.book_details();
    return 0;
}
```

Fill in the blank at `LINE-1` so that the program will print `Modern C++:Partha Pratim Das:IITM`

a) `(const book)this`

b) `(book*)this`

c) `const_cast<book*>(this)`

d) `static_cast<book*>(this)`

Answer: b), c)
Explanation: As object `b` is a constant object, to modify the data-member publisher first the current object needs to be converted to a non-constant object. It can be done at `LINE-1` either of the two ways: `(book*)this`
or
`const_cast<book*>(this)`

Question 8

Consider the code segment given below. *[MCQ, Marks 2]*

```
#include <iostream>
using namespace std;
class Base{
    public:
        virtual void f(){
            cout << "B::f() ";
        }
};
class Derived : public Base{
    public:
        virtual void f(){
            cout << "D::f() ";
        }
};
int main() {
    Base obA;
    Derived obB;
    Base& ra1 = static_cast<Base&>(obB); //LINE-1
    ra1.f();
    Derived& rb1 = static_cast<Derived&>(obA); //LINE-2
    rb1.f();
    Base& ra2 = dynamic_cast<Base&>(obB); //LINE-3
    ra2.f();
    Derived& rb2 = dynamic_cast<Derived&>(obA); //LINE-4
    rb2.f();
    return 0;
}
```

Which line will give you a runtime error?

a) `LINE-1`

b) `LINE-2`

c) `LINE-3`

d) `LINE-4`

Answer: d)
Explanation: The statement at `LINE-1` is having upper-casting, which may be done using static cast. Hence, `ra1.f();` prints `D::f()`.
The statement at `LINE-2`, is having down-casting, which may be done using static cast. Hence, `rb1.f();` prints `B::f()`.
The statement at `LINE-3`, is having upper-casting, which may be done using dynamic cast. Hence, `ra2.f();` prints `D::f()`.
The statement at `LINE-4`, is having down-casting, which cannot be done using dynamic cast. Hence, it generates a run time error.

Question 9

Consider the code segment given below. *[MCQ, Marks 2]*

```
#include <iostream>
using namespace std;
class B{ public: virtual ~B(){} };
class D : public B{};
class DD : public B{};
int main(){
    B objA;
    D objB;
    B* pA = dynamic_cast<B*>(&objB); //LINE-1
    pA == NULL ? cout << "10 " : cout << "11 ";
    D* pB = dynamic_cast<D*>(pA); //LINE-2
    pB == NULL ? cout << "20 " : cout << "21 ";
    DD* pC = dynamic_cast<DD*>(new B); //LINE-3
    pC == NULL ? cout << "30 " : cout << "31 ";
    pC = dynamic_cast<DD*>(&objB); //LINE-4
    pC == NULL ? cout << "40 " : cout << "41 ";
    return 0;
}
```

What will be the output?

a) `11 21 31 40`

b) `11 21 30 40`

c) `11 20 31 40`

d) `11 21 31 41`

Answer: b)
Explanation:
The type-casting at `LINE-1` is valid as it is an upper-casting.
At `LINE-2`, though it is a down-casting, it is allowed as the pointer `pB` points to the same type of object (which of type D).
At `LINE-3`, the down-casting is invalid as the pointer pC points to the parent type of object (which is of type `B`). At `LINE-4`, the casting is also invalid as the pointer pC points to an object (which is of type `D`) that is neither of its base type or derived type.

Programming Questions

Question 1

Consider the following program. Fill in the blanks as per the instructions given below:

- at `LINE-1`, complete the constructor statement,
- at `LINE-2` and `LINE-3`, complete the operator overloading function header,

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

```
#include<iostream>
#include<cctype>
using namespace std;
class Char{
    char ch;
    public:
        _____________ : ch(tolower(_ch)){} //LINE-1
        ___________________{ return ch; } //LINE-2
        ________________{ return ch - 'a' + 1; } //LINE-3
};
int main(){
    char c;
    cin >> c;
    Char cb = c;
    cout << (char)cb << ": position is " << int(cb);
    return 0;
}
```

Public 1

```
Input: c
Output: c: position is 3
```

Public 2

```
Input: G
Output: g: position is 7
```

Private

```
Input: A
Output: a: position is 1
```

Answer:
Answer:
`LINE-1:` `Char(char _ch)`
`LINE-2:` `operator char()`
`LINE-3:` `operator int()`
Explanation:
The statement `Char cb = c;` requires casting from char to `class Char`, which is implemented by the constructor as: `Char(char _ch)`
The statement `(char)cb` requires a casting operator from `class Char` to `char` which may be done by function: `operator char()`

The statement `int(cb)` requires a casting operator from `class Char` to `int` which may be done by function: `operator int()`

Question 2

Consider the following program. Fill in the blanks as per the instructions given below.

- at `LINE-1`, complete the constructor definition,
- at `LINE-2`, complete the overload function header,

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

```
#include<iostream>
#include<cstring>
#include<malloc.h>
using namespace std;
class String{
    char* _str;
    public:
        _________________ : _str(str){} //LINE-1
        ____________________{ //LINE-2
            char* t_str = (char*)malloc(sizeof(_str) + 7);
            strcpy(t_str, "Coding is ");
            strcat(t_str, _str);
            return t_str;
        }
};
int main(){
    char s[20];
    cin >> s;
    String st = static_cast<String>(s);
    cout << static_cast<char*>(st);
    return 0;
}
```

Public 1

```
Input: fun
Output: Coding is fun
```

Public 2

```
Input: easy
Output: Coding is easy
```

Private

```
Input: logical
Output: Coding is logical
```

Answer:
`LINE-1:  String(char* str)`
`LINE-2:  operator char*()`
Explanation:
The constructor at `LINE-1` can be defined as `String(char* str)`. In `LINE-2`, `char*` casting operator is overloaded. It can be done as `operator char*()`.

Question 3

Consider the following program. Fill in the blanks as per the instructions given below:

- at `LINE-1`, complete the overload function header,
- at `LINE-2`, complete the casting operator statement,
- at `LINE-3`, complete the casting operator statement,

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

```
#include<iostream>
using namespace std;
class ClassA{
    int a = 10;
    public:
        void display(){
            cout << a << " ";
        }
};
class ClassB{
    int b = 20;
    public:
        void display(){
            cout << b;
        }
        _________________(int x){ //LINE-1
            b = b + x;
        }
};
void fun(const ClassA &t, int x){
    ClassA &u = ____________________(t); //LINE-2
    u.display();
    ClassB &v = ________________________(u); //LINE-3
    v = x;
    v.display();
}
int main(){
    ClassA t1;
    int a;
    cin >> a;
    fun(t1,a);
    return 0;
}
```

Public 1

```
Input: 5
Output: 10 15
```

Public 2

```
Input: 3
Output: 10 13
```

Private

```
Input: 10
Output: 10 20
```

Answer:

```
LINE-1:  void operator=
LINE-2:  const_cast<ClassA&>
LINE-3:  reinterpret_cast<ClassB&>
```

Explanation:
As per the function `fun()`, we need to overload operator equal to for the class ClassB at `LINE-1` so that the assignment `v = x` will be valid. It can be done as `operator=(int x)`.
To call a non constant function `display(.)` using a const object reference u, we need to cast the reference to a non-const reference. So, `LINE-2` will be filled as `const_cast<ClassA&>`.
Casting between two unrelated classes at `LINE-3` can be done as `reinterpret_cast<ClassB&>`.