Programming in Modern C++: Assignment Week 7

Total Marks : 25

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

February 28, 2025

Question 1

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

```
#include <iostream>
using namespace std;
class Employee {
    string name ;
public:
    Employee(string _name = "unknown") : name(_name) {}
    void update(string na) const{
        ( ____________________________ )->name = na; //LINE-1
    }
    void showInfo() const {
        cout << "Name: " << name;
    }
};
int main(void) {
    const Employee e("Sam");
    e.update("Sameer");
    e.showInfo();
    return 0;
}
```

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

a) `const_cast <Employee*> (this)`

b) `static_cast <Employee*> (this)`

c) `dynamic_cast <Employee*> (this)`

d) `(Employee*)(this)`

Answer: a), d)
Explanation:
The statement `const Employee e("Sam");` defines e as a constant object. To modify its data-members the constant-ness of the object need to be removed. This can be done either by `const_cast` (in option a) or casting constant this pointer to `Employee*` type (in option d).

Question 2

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

```
#include <iostream>
using namespace std;
int incr(int* ptr){
    return ++(*ptr);
}
int main(void) {
    int x = 5;
    const int *ptr = &x;
    x = incr(__________________________); //LINE-1
    cout << x;
    return 0;
}
```

Fill in the blank at `LINE-1` such that the program will print `6`.

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

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

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

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

Answer: a)
Explanation:
The function `incr()` modify the value of `*ptr` and the modified value is returned as return by value. But, in main() function `ptr` is declared as `const int *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 3

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

```
#include<iostream>
using namespace std;
class A1{
    public:
        virtual void f() {}
        void g() {}
};
class A2 : public A1{
    public:
        virtual void g() {}
        void h() {}
        virtual void i();
};
class A3 : public A2{
    public:
        void g() {}
        virtual void h() {}
};
```

What will be the virtual function table for the class A3?

a) `A1::f(A1* const)`
   `A3::g(A3* const)`
   `A3::h(A3* const)`
   `A2::i(A2* const)`

b) `A1::f(A1* const)`
   `A2::g(A2* const)`
   `A3::h(A3* const)`
   `A2::i(A2* const)`

c) `A1::f(A1* const)`
   `A2::g(A2* const)`
   `A2::h(A2* const)`
   `A3::i(A3* const)`

d) `A1::f(A1* const)`
   `A2::g(A2* const)`
   `A3::h(A3* const)`
   `A3::i(A3* const)`

Answer: a)
Explanation:
All four functions are virtual in the class A3. So, there will be four entries in virtual function table.
Now function `f()` is not overridden in class A2 and A3. So, the entry for function `f()` in the virtual function table of class A3 will be `A1::f(A1* const)`.
The function `g()` is virtual from class A2 and is overridden in class A3. So, the entry for function `g()` in VFT of class A3 will be `A3::g(A3* const)`.
The function `h()` is declared as virtual in class A3. So, the entry for function `h()` in VFT of class A3 will be `A3::h(A3* const)`.

Question 4

How many virtual tables will be created for the following classes. *[MCQ, Marks 2]*

```
class A { public: void f() { } };
class B : public A { public: virtual void f() { } };
class C : public A { public: void g() {} };
class D : public B, public C{ public: void g(){ }};
```

a) `1`

b) `2`

c) `3`

d) `4`

Answer: b)
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). Here, only class B and D will have VFTs.

Question 5

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

```
class st1 { };
class st2 { };
st1* s1 = new st1;
st2* s2 = new st2;
```

Which of the following type-casting is permissible?

a) `st2 = static_cast<st2*>(s1);`

b) `st2 = dynamic_cast<st2*>(s1);`

c) `st2 = reinterpret_cast<st2*>(s1);`

d) `st2 = const_cast<st2*>(s1);`

Answer: c)
Explanation:
On each option, there is an attempt to cast from st1* to st2*, 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 c) is correct.

Question 6

Consider the following code segment. *[MSQ, 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());
    return 0;
}
```

What will be the output?

a) `0101`

b) `1010`

c) `0111`

d) `1011`

Answer: a)
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.

Question 7

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

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

What will be the output?

a) `Derived Derived Base`

b) `Derived Derived Base Base`

c) `Derived Derived Base std::bad_cast`

d) `Derived Base Derived Base`

Answer: c)
Explanation:
The statement at `LINE-1`, is having upcasting which may be done using `static_cast`. Hence, `ra1.f();` prints `Derived`.
The statement at `LINE-2`, is having upcasting which may be done using `dynamic_cast`. Hence, `rb1.f();` prints `Derived`.
The statement at `LINE-3`, is having downcasting which may be done using `static_cast`. Hence, `ra2.f();` prints `Base`.

The statement at `LINE-4`, is having downcasting which cannot be done using `dynamic_cast`. Hence, it throws an exception and prints `std::bad cast`.

Question 8

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

```
#include <iostream>
using namespace std;
class A{ public: virtual ~A(){} };
class B : public A{};
class C : public A{};
int main(){
    A objA;
    B objB;
    A* pA = static_cast<A*>(&objB); //LINE-1
    pA == NULL ? cout << "cast-1 invalid:" : cout << "cast-1 valid:";
    B* pB = static_cast<B*>(pA); //LINE-2
    pB == NULL ? cout << "cast-2 invalid:" : cout << "cast-2 valid:";
    C* pC = dynamic_cast<C*>(new A); //LINE-3
    pC == NULL ? cout << "cast-3 invalid:" : cout << "cast-3 valid:";
    pC = dynamic_cast<C*>(&objB); //LINE-4
    pC == NULL ? cout << "cast-4 invalid" : cout << "cast-4 valid";
    return 0;
}
```

What will be the output?

a) `cast-1 valid:cast-2 valid:cast-3 invalid:cast-4 invalid`

b) `cast-1 valid:cast-2 invalid:cast-3 invalid:cast-4 invalid`

c) `cast-1 valid:cast-2 valid:cast-3 valid:cast-4 invalid`

d) `cast-1 valid:cast-2 invalid:cast-3 valid:cast-4 invalid`

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

Question 9

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

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

class B{
    int a;
    public:
        B(int x) : a(x) {}
        int get() { return a; }
        void set(int x) { a = x + 1; }
};

class D : public B{
    int b;
    public:
        D(int x, int y) : B(x), b(y) {}
        void change(int c){
            static_cast<B>(*this).set(c);
            b = c+1;
        }
        void print(){
            cout << B::get() << " " << b;
        }
};

int main() {
    D d(10,20);
    d.change(7);
    d.print();
    return 0;
}
```

What will be the output?

a) `10 8`

b) `10 7`

c) `7 21`

d) `7 20`

Answer: a)
Explanation:
As per the `change()` function definition, `static_cast` operator will create a temporary object of `B` class type and assign its data member with the value $7 + 1 = 8$. But, actual `B` class object data member is not changed. However, the change of `D` class data member is straightforward. Hence, the program will print `10 8`.

Programming Questions

Question 1

Complete the program with the following instructions.

- Fill in the blank at `LINE-1` to complete the parameterized constructor definition,
- Fill in the blanks at `LINE-2` to complete cast operator overloading function to typecast to `IP1` type.

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

```
#include <iostream>
using namespace std;
class IP1 {
    int i;
    public:
        IP1(int ai) : i(ai) {}
        int get() const { return i; }
        void update() { i *= 10; }
    };
    class IP2 {
        int i;
        public:
            IP2(int ai) : i(ai) {}
            int get() const { return i; }
            ---------------------------------- //LINE-1
            ---------------------------------- //LINE-2
            void update() { i *= 20; }
    };
int main() {
    int i;
    cin >> i;
    IP1 a(i+2);
    IP2 b(i);
    const IP2 &r = static_cast<IP2>(a);
    a.update();
    cout << a.get() << ":";
    cout << r.get() << ":";
    const IP1 &s = static_cast<IP1>(b);
    b.update();
    cout << b.get() << ":";
    cout << s.get() << ":";
    return 0;
}
```

Public 1

```
Input: 2
Output: 40:4:40:2:
```

Public 2

```
Input: 5
Output: 70:7:100:5:
```

Private 1

```
Input: 15
Output: 170:17:300:15:
```

Answer:

```
LINE-1:  IP2(IP1& a) :  i(a.get()) {}
LINE-2:  operator IP1() { return IP1(i); }
```

Explanation:

`static_cast` can explicitly call a single-argument constructor or a conversion operator (that is, User-Defined Cast) to handle the casting between two unrelated classes. Here both are present.

Question 2

Consider the following program with the following instructions.

- Fill in the blank at `LINE-1` to complete assignment operator overload function header.
- Fill in the blanks at `LINE-2` and `LINE-3` to complete statements with the appropriate casting operator.

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

```
#include<iostream>
using namespace std;
class myClassA{
    int a = 5;
    public:
        void print(){
            cout << a << " ";
        }
};
class myClassB{
    int b = 10;
    public:
        void print(){
            cout << b;
        }
        ____________________{ //LINE-1
            b = b * x;
        }
};
void fun(const myClassA &t){
    int x;
    cin >> x;
    myClassA &u = ____________________________(t); //LINE-2
    u.print();
    myClassB &v = ______________________________(u); //LINE-3
    v = x;
    v.print();
}
int main(){
    myClassA t1;
    fun(t1);
    return 0;
}
```

Public 1

```
Input: 5
Output: 5 25
```

Public 2

```
Input: 7
Output: 5 35
```

Private

```
Input: 100
Output: 5 500
```

Answer:

```
LINE-1:  void operator=(int x)
LINE-2:  const_cast<myClassA&>
LINE-3:  reinterpret_cast<myClassB&>
```

Explanation:
As per the function `fun()`, we need to overload assignment operator for the class `myClassB` 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 `print(.)` using a constant object reference `u`, we need to cast the reference to a non-const reference. So, `LINE-2` will be filled as `const_cast<myClassA&>`.
Casting between two unrelated classes at `LINE-3` can be done as `reinterpret_cast<myClassB&>`.

Question 3

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

- at `LINE-1` to complete the constructor definition,
- at `LINE-2` to complete `char*` operator 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, "Welcome ");
        strcat(t_str, _str);
        return t_str;
    }
};
int main(){
    char s[15];
    cin >> s;
    String st = static_cast<String>(s);
    cout << static_cast<char*>(st);
    return 0;
}
```

Public 1

```
Input: Amal
Output: Welcome Amal
```

Public 2

```
Input: Som
Output: Welcome Som
```

Private

```
Input: Sir
Output: Welcome Sir
```

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. In can be done as `operator char*()`.