# Programming in 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 27, 2020

## Question 1

Consider the program below. *[MSQ, Marks 2]*

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

class employee {
    int emp_id;
    string name;
public:
    employee(int _emp_id, string _name) : emp_id(_emp_id), name(_name) { }

    void update(int id, string na) const {
        (____________________)->emp_id = id;   // LINE-1
        (____________________)->name = na;     // LINE-2
    }

    void showInfo() const {
        cout << emp_id << " : " << name;
    }
};

int main() {
    const employee e(3, "Raj");

    e.update(30, "Rajan");
    e.showInfo();

    return 0;
}
```

Fill in the blank at `LINE-1` and `LINE-2` such that the output is:

```
30 :  Rajan
```

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(3, "Raj");` 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 program below. *[MCQ, Marks 2]*

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

class Base {
    int data;
public:
    Base() { cout << "Base()" << " "; }
    Base(int _x) { cout << "Base(x)" << " "; }
};

class Derived1 : virtual public Base {
public:
    Derived1() { cout << "Derived1()" << " "; }
    Derived1(int _x) : Base(_x) { cout << "Derived1(x)" << " "; }
};

class Derived2 : virtual public Base {
public:
    Derived2() { cout << "Derived2()" << " "; }
    Derived2(int _x) : Base(_x) { cout << "Derived2(x)" << " "; }
};

class ReDerived : public Derived1, public Derived2 {
public:
    ReDerived() { cout << "ReDerived()" << " "; }
    ReDerived(int x) : Derived1(x), Derived2(x) { cout << "ReDerived(x)" << " "; }
};

int main() {
    ReDerived(5);

    return 0;
}
```

How many virtual table will be set up by the compiler?

a) `Base(x) Derived1(x) Base(x) Derived2(x) ReDerived(x)`

b) `Base() Derived1(x) Base() Derived2(x) ReDerived(x)`

c) `Base() Derived1(x) Derived2(x) ReDerived(x)`

d) `Base() Base(x) Derived1(x) Derived2(x) ReDerived(x)`

**Answer**: c)
**Explanation:**
Class `ReDerived` is inherited from classes `Derived1`, and `Derived2`. Hence, the constructor of `ReDerived` will call constructors of `Derived1` and `Derived2` in that order (note that, the constructors are actually called in the code as `Derived1(x), Derived2(x)` in the initialization list, however, the order in which they are defined for inheritance will prevail).
First, `Derived1` is virtually inherited from `Base` (and no `Base` object has been constructed so far

for `ReDerived`), so first the default constructor of `Base` will be called followed by parameterized constructor of `Derived1`.

Next, `Derived2` is also virtually inherited from `Base` and an `Base` object has been already been constructed for `ReDerived` (during construction of `Derived1` object), so parameterized constructor of `Derived2` will be directly called (no call of any constructor of `Base`).

## Question 3

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

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

int incr(int* ptr) {
    return (*ptr)++;
}

int main() {
    int val = 10;
    const int *ptr = &val;

    val = incr(____________________);    // LINE-1
    cout << val;

    return 0;
}
```

Fill in the blank at `LINE-1` so that it will print 10.

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` but the previous 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 4

Consider the below class hierarchy. *[MCQ, Mark 2]*

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

class A {
public:
    virtual void f() { }
    void g() {}
};

class B : public A {
public:
    virtual void g() { }
    void h() { }
    virtual void i();
};

class C : public B {
public:
    void g() { }
    virtual void h() { }
};
```

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

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

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

**Answer**: a)
**Explanation:**
All four functions are virtual in the class `C`. So, there will be four entries in virtual function table.
Now function `f()` is not overridden in class `B` and `C`. So, the entry for function `f()` in the virtual function table of class `C` will be `A::f(A* const)`.
The function `g()` is virtual from class `B` and is overridden in class `C`. So, the entry for function

`g()` in VFT of class `C` will be `C::g(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 5

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

```
class A { public: virtual void f() { } };
class B : public A { };
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**: 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 three classes, thus, will setup virtual function tables.

## Question 6

Which statement in the following program generates compilation error? *[MSQ, Marks 2]*

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

int main() {
    int i = 10;
    double d = 3.14;
    int *ip = &i;
    double *pd;

    i = static_cast<int>(d);           // statement-1
    d = static_cast<double>(i);        // statement-2
    pd = static_cast<double*>(ip);     // statement-3
    i = static_cast<int>(&i);          // statement-4

    return 0;
}
```

a) `statement-1`

b) `statement-2`

c) `statement-3`

d) `statement-4`

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

## Question 7

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

```
struct st1 { };
struct 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 structures 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 8

Consider the below classes. *[MSQ, Mark 2]*

```
class Base {
    public:
        void f() { }
};

class Derived : public Base {
    public:
        void g() { }
};
```

What is/are the appropriate option/s to perform up-casting where

```
    Base b;
    Derived d;
```

a) `Base *bp = &d;`

b) `Base *bp = (Base*)&d;`

c) `Derived *dp = &b;`

d) `Derived *dp = (Derived*)&d;`

**Answer**: a), b)
**Explanation:**
A pointer type can point to its own type or its derived type but cannot point to its base type. In option a) or b), the derived class pointer `&d`, is assigned to base class object pointer `bp` is a correct form of up-casting. (c) is down-casting which is invalid in C++ and (d) is a casting to the same type.

## Question 9

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

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

class Parent { virtual void fun() { } };
class Child : public Parent { void fun() { } };

int main() {
    try {
        Parent *pbd = new Child;
        Parent *pbb = new Parent;

        Child *pd = dynamic_cast<Child*>(pbd); // LINE-1
        if (pd == 0)
            cout << "Null pointer on first type-cast" << endl;

        pd = dynamic_cast<Child*>(pbb); // LINE-2
        if (pd == 0)
            cout << "Null pointer on second type-cast" << endl;

        pd = static_cast<Child*>(pbd); // LINE-3
        if (pd == 0)
            cout << "Null pointer on third type-cast" << endl;
    }
    catch (exception& e) {
        cout << "Exception: " << e.what();
    }

    return 0;
}
```

What will be the output?

a) `Null pointer on first type-cast`

b) `Null pointer on third type-cast`

c) `Exception:  NULL pointer exception`

d) `Null pointer on second type-cast`

**Answer**: d)
**Explanation:**
In `LINE-1`, we polymorphically cast `pbd`, a pointer of `Parent` type and actually pointing to an object of `Child` type, to pointer to `Child` type. Hence, it is valid.
In contrast, in `LINE-2`, we polymorphically cast `pbb`, a pointer of `Parent` type and actually pointing to an object also of `Parent` type, to pointer to `Child` type. This cannot be done. It is a down-casting. Hence `pd` will be `NULL`.
Also, no exception is raised as the casting is done in pointers.
In `LINE-3`, we non-polymorphically cast `pbb`, a pointer of `Parent` type and actually pointing to an object also of `Parent` type, to pointer to `Child` type. Although it is a down-casting, but

can be done with `static_cast`. Hence `pd` will not be `NULL`.

## Programming Questions

## Question 1

Consider the following program. Fill in the blanks at `LINE-1` (Casting using constructor) and `LINE-2` (User defined cast using operator function) with appropriate expression for casting between two unrelated classes such that it would satisfy the given test cases. *Marks: 3*

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

class A {
    int i;
public:
    A(int ai) : i(ai) {}
    int get() const { return i; }
    void update() { i *= 10; }
};

class B {
    int i;
public:
    B(int ai) : i(ai) {}
    int get() const { return i; }
    -------------------- // LINE-1
    -------------------- // LINE-2
    void update() { i *= 20; }
};

int main() {
    int i;
    cin >> i;

    A a(i++);
    B b(i);

    const B &r = static_cast<B>(a);
    a.update();
    cout << a.get() << ":";
    cout << r.get() << ":";

    const A &s = static_cast<A>(b);
    b.update();
    cout << b.get() << ":";
    cout << s.get() << ":";

    return 0;
}
```

### Public 1

```
Input: 1
Output: 10:1:40:2:
```

### Public 2

```
Input: 15
Output: 150:15:320:16:
```

### Private

```
Input: 10
Output: 100:10:220:11:
```

**Answer:**

LINE-1: `B(A& a) : i(a.get()) {}`

LINE-2: `operator A() { return A(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. Fill in the blanks at LINE-1 with appropriate abstract function definition for function `fun()`, and LINE-2, LINE-3 and LINE-4 with appropriate inheritance such that it matches the given test cases. *Marks: 3*

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

class A {
protected:
    int i;
public:
    A(int a = 0) : i(a) { }
    void set(int a) {
        i = a;
        cout << i << " ";
    }
    // create abstract function fun()
    ____________________ // LINE-1
};

class B : __________ {  // LINE-2
public:
    B(int a = 0) : A(a) { }
    void fun() { set(i * 10); }
};

class C : __________ {  // LINE-3
public:
    C(int a = 0) : A(a) { }
    void fun() { set(i + 10); }
};

class D : __________ {  // LINE-4
public:
    D(int a = 0) : A(a) { }
    void fun() { set(i - 10); }
};

int main() {
    int num;
    cin >> num;

    A *pt[3] = { new B(num), new C(num), new D(num) };
    for (int i = 0; i < 3; i++) {
        pt[i]->fun();
    }

    return 0;
}
```

### Public 1

```
Input:  5
Output:  50 15 -5
```

### Public 2

```
Input:  -20
Output:  -200 -10 -30
```

### Private

```
Input:  10
Output:  100 20 0
```

**Answer:**

```
LINE-1:  virtual void fun() = 0;
LINE-2:  public A
LINE-3:  public A
LINE-4:  public A
```

**Explanation**:
The abstract function at LINE-1 must be filled by `virtual void fun() = 0;`.
The LINE-2, LINE-3, and LINE-4 must be filled by `public A`.

## Question 3

Consider the following program. Fill in the blank at LINE-1, LINE-2, and LINE-3 with appropriate inheritance type such that it satisfies the given test cases.

*Marks: 3*

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

class A {
public:
    A(int i = 0) { cout << i << " "; }
    void print(int i){ cout << i << " "; }
};

class B : ____________________ { // LINE-1 : Inherit from class A
public:
    B(int i = 0) : A(++i) { print(i); }
};

class C : ____________________ { // LINE-2 : Inherit from class A
public:
    C(int i = 0) : A(++i) { print(i); }
};

class D : ____________________ { // LINE-3 : Inherit from class B and C
public:
    D(int i = 0) : B(i * 2), C(++i) { print(i); }
};

int main() {
    int i;
    cin >> i;

    D obj(i);

    return 0;
}
```

### Public 1

```
Input:10
Output: 0 11 22 11
```

### Public 2

```
Input: 5
Output: 0 6 12 6
```

### Private

```
Input: 15
Output: 0 16 32 16
```

**Answer:**
LINE-1: `virtual public A`
LINE-2: `virtual public A`
LINE-3: `public C, public B`
**Explanation**:
From the all test-cases, we can see that the default constructor for the class `A` is called only once. Hence, the classes `B` and `C` used virtual inheritance. So the `LINE-1` and `LINE-2` need to be filled by `virtual public A`.
Class `D` should be inherited from class `B` and `C`. From class `B` and `C`, `class C` is invoked before `class B`. Hence the LINE-3 need to be filled by `public C, public B`.

## Question 4

Consider the following program. Fill in the blank at LINE-1 with appropriate initializer statement for the class constructor. Fill in the blanks at LINE-2 and LINE-3 with appropriate function headers for operator overloading such that it satisfies the given test cases. *Marks: 3*

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

class Container {
    int i;
    int *arr;
public:
    Container(int k) : ____________________ { } // LINE-1

    ____________________ { // LINE-2
        return arr[--i];
    }

    ____________________ { // LINE-3
        int t;
        for (int j = 0; j < k; j++){
            cin >> t;
            this->arr[j] = t;
        }
        return *this;
    }
};

int main() {
    int k;
    cin >> k;

    Container c(k);
    c = k;

    for (int i = 0; i < k; i++)
        cout << static_cast<int>(c) << " ";

    return 0;
}
```

### Public 1

```
Input: 4
10 20 30 40
Output:
40 30 20 10
```

### Public 2

```
Input: 6
23 45 23 10 -3 -10
Output:
-10 -3 10 23 45 23
```

### Private

```
Input: 5
9 1 2 3 5
Output:
5 3 2 1 9
```

**Answer:**
LINE-1: `i(k), arr(new int[i])`
LINE-2: `operator int()`
LINE-3: `Container operator=(int& k)`
**Explanation**:
The initialization of the data-members at LINE-1 can be done as:
`i(k), arr(new int[i])`. Note the use of array allocation.
At LINE-2, we overload type-casting operator for the statement `static_cast<int>(c)` as:
`operator int()`
At LINE-3, we overload `operator=` for the statement `c = k;` as:
`Container operator=(int& k)`. Note that `k` may be used as value parameter too.