TERM	COURSE NAME	COURSE CODE	VERSION
Fall-2019-Quiz-Lecture-2	Object-Oriented Software Development using C++	OOP345	A

1. Problems that arise with dynamic typing include:
    a. determining the dynamic type in copying a polymorphic object to another polymorphic object
    b. specializing an operation for a dynamic type
    c. excluding a specific type from most derived selection
    d. All of the above
    e. None of the above
2. Copying of a polymorphic object at different stages of execution requires knowledge of its:
    a. Dynamic type
    b. Static type
    c. All of the above
    d. None of the above
3. Why cloning of a polymorphic object at different stages of execution is not straight forward and requires extra information?
    a. In order to allocate the correct amount of memory for the copy
    b. In order to delete the first object
    c. In order to move the second object
    d. All of the above
    e. None of the above
4. To determine the dynamic type at run-time we can define a cloning member function for each concrete class in the hierarchy
    a. YES
    b. NO

Code1.0

```
#ifndef SHAPE_H
#define SHAPE_H
// Polymorphic Objects - Cloning
// Shape.h

class Shape {
public:
    virtual double volume() const = 0;
    virtual Shape* clone()  const = 0;
};
#endif
```

```
// Polymorphic Objects - Cloning
// Cube.h

#include "Shape.h"

class Cube : public Shape {
    double len;
public:
    Cube(double);
    double volume() const;
    Shape* clone() const;
};
```

```
// Polymorphic Objects - Cloning
// Sphere.h

#include "Shape.h"

class Sphere : public Shape {
    double rad;
public:
    Sphere(double);
    double volume() const;
    Shape* clone() const;
};
```

```
// Polymorphic Objects - Cloning
// Cube.cpp

#include "Cube.h"

Cube::Cube(double l) : len(l) {}

Shape* Cube::clone() const {
    return new Cube(*this);
}

double Cube::volume() const {
    return len * len * len;
}
```

```
// Polymorphic Objects - Cloning
// Sphere.cpp

#include "Sphere.h"

Sphere::Sphere(double r) : rad(r) {}

Shape* Sphere::clone() const {
    return new Sphere(*this);
}

double Sphere::volume() const {
    return 4.18879 * rad * rad * rad;
}
```

```
// Polymorphic Objects - Cloning
// cloning.cpp

#include <iostream>
#include "Cube.h"
#include "Sphere.h"

void displayVolume(const Shape* shape) {
    if (shape)
        std::cout << shape->volume() << std::endl;
    else
        std::cout << "error" << std::endl;
}

Shape* select() {
    Shape* shape;
    double x;
    char c;
    std::cout << "s (sphere), c (cube) : ";
    std::cin >> c;
    if (c == 's') {
        std::cout << "dimension : ";
        std::cin >> x;
        shape = new Sphere(x);
    } else if (c == 'c') {
        std::cout << "dimension : ";
        std::cin >> x;
        shape = new Cube(x);
    } else
        shape = nullptr;
    return shape;
}

int main() {
  1. Shape* shape = select();
  2. Shape* clone = shape->clone();
  3. displayVolume(shape);
  4. displayVolume(clone);
  5. delete clone;
  6. delete shape;
}
```

5. First run of Code 1.0, the user selects ‘s’, and dimension of ‘2’, therefore the output of line 3 is:
    a. 4.18879
    b. 2
    c. 33.1503
    d. All of the above
    e. None of the above
6. First run of Code 1.0, the user selects ‘s’, and dimension of ‘2’, therefore the output of line 4 is:
    a. 4.18879
    b. 2
    c. 33.1503
    d. All of the above
    e. None of the above

7. Referencing Q6&7, does the variable ‘shape’. Have the same value as the variable ‘clone’:
   a. YES
   b. NO

Code2.0

Main.cpp

```cpp
1.  #include <iostream>
2.  #include <exception>
3.  using namespace std;
4.  class Base { virtual void dummy() {} };
5.  class Derived: public Base { int a; };
6.  class DerivedSecond: public Base { int b;};
7.  int main () {
8.      try {
9.              Base * pba = new Derived;
10.             Base * pbc = new DerivedSecond;
11.             Base * pbb = new Base;

12.             Derived * pd;
13.             Base * pbase;

14.             pd = dynamic_cast<Derived*>(pba);
15.             if (pd==0) cout << "Null pointer on first type-cast.\n";

16.             pd = dynamic_cast<Derived*>(pbc);
17.             if (pd==0) cout << "Null pointer on second type-cast.\n";

18.             pd = dynamic_cast<Derived*>(pbb);
19.             if (pd==0) cout << "Null pointer on third type-cast.\n";

20.             pbase = dynamic_cast<Base*>(pba);
21.             if (pbase==0) cout << "Null pointer on fourth type-cast.\n";
22.

23.     } catch (exception& e) {cout << "Exception: " << e.what();}
24.     return 0;}
```

8. In Code 2.0, Line 15 will print “Null pointer on first type-cast”:
   a. YES
   b. NO
9. In Code 2.0, Line 17 will print “Null pointer on second type-cast”:
   a. YES
   b. NO
10. In Code 2.0, Line 19 will print “Null pointer on third type-cast”:
    a. YES
    b. NO
11. In Code 2.0, Line 21 will print “Null pointer on fourth type-cast”:
    a. NO
    b. YES

Code3.0

Main.cpp

```cpp
// Polymorphic Objects - RTTI
 // rtti.cpp

#include <typeinfo> // for typeid
#include <iostream>
class A {
     int x;
     public:
     A(int a) : x(a) {}
     virtual void display() const {
         std::cout << x << std::endl;
     }
};
class B : public A {
    int y;
    public:
    B(int a = 5, int b = 6) : A(a), y(b) {}
    void display() const {
        A::display();
        std::cout << y << std::endl; }
};
class C : public B {
     int z;
     public:
     C(int a = 4, int b = 6, int c = 7) : B(a, b), z(c) {}
     void display() const {
         B::display();
         std::cout << z << std::endl; }
};
// show calls display() on all types except C
//
void show(const A* a) {
     C cref;
     if (typeid(*a) != typeid(cref)) {
        a->display();
     } else   std::cout << typeid(cref).name()
              << " objects are private" << std::endl;
}
int main() {
    A* a[3];
    a[0] = new A(3);
    a[1] = new B(2, 5);
    a[2] = new C(4, 6, 7);
    for(int i = 0; i < 3; i++)
        show(a[i]);
    for(int i = 0; i < 3; i++)
        std::cout << typeid(a[i]).name() << std::endl;
    for(int i = 0; i < 3; i++)
        delete a[i];
}
```

12. In Code 3.0, First iteration of line 42 will print:
    a. 3
    b. 2
       5
    c. Class C Object are private
    d. All of the above
    e. None of the above
13. In Code 3.0, Second iteration of line 42 will print:
    a. 3
    b. 2
       5
    c. Class C Object are private
    d. All of the above
    e. None of the above
14. In Code 3.0, third iteration of line 42 will print:
    a. 3
    b. 2
       5
    c. Class C Object are private
    d. All of the above
    e. None of the above
15. In Code 3.0, First iteration of line 44 will print:
    a. Pointer to type A
    b. Pointer to type B
    c. Pointer to type C
    d. All of the above
    e. None of the above
16. In Code 3.0, Second iteration of line 44 will print:
    a. Pointer to type A
    b. Pointer to type B
    c. Pointer to type C
    d. All of the above
    e. None of the above
17. In Code 3.0, Third iteration of line 44 will print:
    a. Pointer to type A
    b. Pointer to type B
    c. Pointer to type C
    d. All of the above
    e. None of the above

Code4.0

Main.cpp

```cpp
1.  #include <iostream>
2.  #include "cArray.h"
3.  int main() {
4.      Array<> s, t;
5.      Array<int, 50> a, b;
6.      Array<double> u, z;
7.      Array<int, 40> v;
8.      std::cout << Array<>::cnt() << std::endl;
9.      std::cout << Array<double, 50>::cnt() << std::endl;
10.     std::cout << Array<int, 40>::cnt() << std::endl;
11.     std::cout << Array<double>::cnt() << std::endl;
12.     std::cout << Array<int, 50>::cnt() << std::endl;
13. }
```

array.h

```cpp
template <typename T= int, int size = 50>
class Array {
    T a[size];
    unsigned n;
    T dummy;
    static unsigned count;
    public:
        Array() : n{0}, dummy{0} { ++count; }
        T& operator[](unsigned i) {
              return i < 50u ? a[i] : dummy;
        }
        static unsigned cnt() { return count; }
        ~Array() { --count; }
};
template <typename T, int size>
unsigned Array<T, size>::count = 0u;
```

18. In Code 4.0, The output of line 8 is:
    a. 2
    b. 4
    c. 1
    d. 3
    e. None of the above
19. In Code 4.0, The output of line 9 is:
    a. 2
    b. 4
    c. 1
    d. 3
    e. None of the above
20. In Code 4.0, The output of line 10 is:
    a. 2
    b. 4
    c. 1
    d. 3
    e. None of the above
21. In Code 4.0, The output of line 11 is:
    a. 2
    b. 4
    c. 1
    d. 3
    e. None of the above
22. In Code 4.0, The output of line 12 is:
    a. 2
    b. 4
    c. 1
    d. 3
    e. None of the above

Code5.0

Main.cpp

```
1. #include <iostream>
2. using namespace std;
3. template<class T> void f(T x, T y) { cout << " A-A" << endl; }
4. template<class T, class V> void f(T x, V y) { cout << " A-B" << endl; }
5. template<class T, class V, class D> void f(T x, V y, D z) { cout << " A-C" <<
   endl; }
6. void f(int w, int z) { cout << " C-C" << endl; }
7. void f(int w, double z) { cout << " C-D" << endl; }
8. int main() {
9.       f( 1 ,  2 );
10.      f('a', 'b');
11.      f( 1 , 3.5);
12.      f( 3.5  , 1);
13.}
```

23. In Code 5.0, The output of line 9 is:
    a. C-C
    b. A-A
    c. C-D
    d. A-B
    e. None of the above
24. In Code 5.0, The output of line 10 is:
    a. C-C
    b. A-A
    c. C-D
    d. A-B
    e. None of the above
25. In Code 5.0, The output of line 11 is:
    a. C-C
    b. A-A
    c. C-D
    d. A-B
    e. None of the above
26. In Code 5.0, The output of line 12 is:
    a. C-C
    b. A-A
    c. C-D
    d. A-B
    e. None of the above