COSC1076 | Advanced Programming Techniques

Revision Questions | Week 10 | Answers

1. (a) The type of every name entity is fixed and cannot be changed once instantiated
   (b) The type of every name entity may be modified after it is instantiated
   (c) The type of every name entity is determined at compile-time
   (d) The type of every name entity is determined at run-time
2. Types listed in comments below:

```cpp
#include <string>

int main(void) {
    // int
    auto a = -8;

    // double (not float)
    auto b = 6.8;

    // double (not float)
    auto c = 10.0;

    // int*
    auto d = &a;

    // double*
    auto e = &c;

    // const char*
    auto f = "hello world";

    // std::string
    auto g = std::string("hello world");
}
```

3. The type of the new overloaded version of the function must be different from all the type of all other versons of the function. In C++ the type of the function is determined by its parameters.
4. Both define versions of a method with the same name as an existing method. *Overloading* is defining a new separate version with a different type. *Overriding* effectively replaces the existing version of the method.
5. A number of classes, n, where $n \geq 1$, provided that a circular inheritance is not created.
6. (a) Listed in comments below:

```cpp
// 1. Constructor of A
A* a1 = new A();

// 1. Constructor of A
// 2. Constructor of B
B* b1 = new B();

// 1. Constructor of A
// 2. Constructor of B
// 3. Constructor of C
C* c1 = new C();
```

   (b) Listed in comments below:

```
A* a2;
B* b2;
C* c2;

// Yes - classes are the same
a2 = a1;

// Yes - class C is a sub-class of A
a2 = c1;

// NO - class A is NOT a sub-class of B
b2 = a1;

// Yes - class C is a sub-class of B
b2 = c1;

// NO - class A is NOT a sub-class of C
c2 = a1;

// NO - class B is NOT a sub-class of C
c2 = b2;
```

(c) Listed in comments below:

```
// C::foo();
c1->foo();

// A::foo();
a1->foo();

A* a3 = b1;
// B::foo(), under polymorphism
a3->foo();
```

(d) Listed in comments below:

```
// 1. Deconstructor of C
// 2. Deconstructor of B
// 3. Deconstructor of A
delete c1;
```

7. For the function, consider two cases, (1) `T` is a primitive type, and (2) `T` is a class.
   (a) For a primitive types, `T` must support the being added to an integer (line 4).
   (b) For a class, `T` must have:
      i. A Copy constructor, to be passed to the function (line 2), to be copied to a local variable (line 3), and to be copied on being return (line 5).
      ii. Overload the plus (+) operator (line 4).
   (c) As a side note, `T` may also be a pointer, however, there are no additional requirements on the pointer type, since the operations act directly on the pointer, not the type of the dereferenced pointer.

```
template <typename T>
T bar(T value) {
    T retVal = value;
    retVal = retVal + 1;
    return retVal;
}
```

8. With *polymorphism* the type of the object is reinterpreted as being of a different type, however, the type

of the underlying object remains the same. With *typecasting* the type of the object is actually modified.

9. A `static_cast` performs a typecast with compile type checking. A `dynamic_cast` is a typecast conducted at runtime, where a nullptr is returned (for pointer type-casts) if a down-cast cast is not possible.
10. Listed in comments below:

```
// C::foo(), as the pointer type is cast, but no the type of the underlying
↪  object. Therefore through polymorphism the type of c1 is reinterpreted, but
↪  the C version of foo is called.
A* a4 = dynamic_cast<A*>(c1);
a4->foo();

// A::foo(), as the type of the object itself is modified
A a5 = dynamic_cast<A>(*c1);
a4->foo();

// Results in a null pointer exception, as A* pointer to an object of type A
↪  cannot be down-cast.
C* c6 = dynamic_cast<C*>(a1);
c6->foo();
```