Module IN3013/INM173 – Object-Oriented Programming in C++

Solutions to Exercise Sheet 6

1. A suitable `main` function is

```
int main() {
        Cat cat1("miffy");
        Dog dog1("rover");
        Dalmatian dog2("bill", 101);
        cat1.speak();
        dog1.speak();
        speakTwice(dog2);
        return 0;
}
```

   Note that `speakTwice` calls `speak`, which is dynamically bound to the `Dog` version.

2. Firstly, we replace the definition of the method in the class with a declaration:

```
class Dog : public Pet {
public:
        Dog(string name) : Pet(name) {}
        string sound() const { return "woof"; }
        virtual void speak() const;
};
```

   Note that the `const`, indicating that the method do not alter the object, is part of the method's signature. Now we can define the method outside the class, by qualifying it with the class name `Dog`:

```
void Dog::speak() const {
        Pet::speak();
        cout << '(' << _name << " wags tail)\n";
}
```

   Note that we cannot say `virtual` here, though the method is virtual because of its declaration in the class.

   Note also that this is *not* a pure virtual method: `speak` still has a definition for `Dog`, it's just defined outside the class.

3. If we add the line

```
Pet pet("norbert");
```

   we get the error message

```
pets.cc: In function ‘int main()’:
pets.cc:45: cannot declare variable ‘pet’ to be of type ‘Pet’
pets.cc:45:   since the following virtual functions are abstract:
pets.cc:10:    class string Pet::sound() const
```

   The compiler is saying that no objects of type `Pet` can be created, because the method `sound()` has no definition (i.e. it is abstract). In Java, we would have had to declare `sound()` as `abstract`, and therefore the whole class as `abstract`. C++ has no such keyword, but the underlying concept is the same: no objects of the clas can be created, because they would lack some methods.

4. We can define a version that calls the parent method (the equivalent of `super.speak()` in Java):

```
virtual void speak() const {
        Dog::speak();
        cout << ’(’ << _name << " looks cute)\n";
}
```

   We can also call the `Pet` version directly:

```
virtual void speak() const {
        Pet::speak();
        cout << ’(’ << _name << " looks cute)\n";
}
```

   Note that the `virtual` qualifier is not necessary here (or in `Dog`): once a method is declared `virtual`, any overriding is also `virtual`.

5. Assuming the declarations above, if we write

```
dog1 = dog2;
dog1.speak();
```

   then the assignment on the first line *slices* the `Dalmatian` object to make it fit into a `Dog` object, copying only the `Dog` fields. Moreover, the object left of `dog1` is a `Dog`, as may be verified by calling its `speak()` method.

6. A vector of pointers to pets:

```
vector<Pet *> pets;
```

Adding the address of a local `Cat` to the vector:

```
Cat felix("Felix");
pets.push_back(&felix);
```

Adding a pointer to a dynamically allocated `Dog` to the vector:

```
Dog *dog_ptr = new Dog("Fido");
pets.push_back(dog_ptr);
```

or equivalently,

```
pets.push_back(new Dog("Fido"));
```

When we access elements of the vector, dynamic binding is used:

```
for (int i = 0; i < pets.size(); i++)
    pet[i]->speak(); // miao, woof
```

7. Starting with a list of pointers

```
list<Pet *> pets;
```

The associated iterator type returns pointers:

```
typedef list<Pet *>::iterator Iter;
for (Iter p = pets.begin(); p != pets.end(); ++p)
    (*p)->speak();
```

In the last line,

- `p` is an iterator, and `*p` uses an overloaded definition of the `*` operator.
- The value of `*p` is an element of the container, in this case a pointer to `Pet`, so here `->` is the built-in pointer dereferencing.