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

Solutions to Exercise Sheet 8

1. (a) A suitable class is

```
class A {
public:
        A() { cout << "creating an A\n"; }
        virtual ~A() { cout << "destroying an A\n"; }
};
```

We can test it (in a `main` function) by creating a local object and a dynamically allocated one:

```
        {
                A a;
                cout << "using a\n";
        }
        A *ap;
        ap = new A();
        cout << "using *ap\n";
        delete ap;
```

The locally allocated one is destroyed at the end of the block in which it is declared, so the output is

```
creating an A
using a
destroying an A
creating an A
using *ap
destroying an A
```

(b) We add a field to record to each object to record how many times the destructor for this object has run. (If it's more than 1, something has gone wrong.)

```
class A {
        int destroy_count;
public:
        A() : destroy_count(0) { cout << "creating an A\n"; }
        virtual ~A() {
                destroy_count++;
```

```
                cout << "destroying an A (" << destroy_count << ")\n";
        }
};
```

(c) Our class `B` is

```
class B {
        A a;
public:
        B() { cout << "creating a B\n"; }
        virtual ~B() { cout << "destroying a B\n"; }
};
```

Now if we test this with local allocation:

```
        {
                B b;
                cout << "using b\n";
        }
```

we get the output

```
creating an A
creating a B
using b
destroying a B
destroying an A (1)
```

Note that the filed `a` of type `A` is initialized before the body of the `B` constructor is run, and that similarly it is destroyed after the body of the destructor has run.

(d) Switching to a dynamically allocated field:

```
class B {
        A *ap;
public:
        B() : ap(new A()) { cout << "creating a B\n"; }
        virtual ~B() {
                cout << "destroying a B\n";
                delete ap;
        }
};
```

Note the initialization of the pointer field `ap` in the constructor. Thus we need to destroy the dynamically allocated object using `delete` in the destructor.

(e) For the copy constructor, we create a new `A` object, initialized as a copy of the other `A` object, namely `*other.ap`.

```
B(const B &other) : ap(new A(*other.ap)) {}
```

This uses the automatically provided copy constructor of the `A` class, which simply copies across all fields. This fine for the `A` class, but not `B`.

The general form of the assignment operator is

```
const B & operator=(const B &other) {
        if (&other != this) {
                // ...
        }
        return *this;
}
```

(This is placed inside the `B` class.) We do nothing when an object is assigned to itself, and we always return the target of the assignment, `*this`. A simple recipe for the bit in the middle is to do the effect of the destructor

```
delete ap;
```

followed by the effect of the constructor:

```
ap = new A(*other.ap);
```

giving the following assignment operator:

```
const B & operator=(const B &other) {
        if (&other != this) {
                delete ap;
                ap = new A(*other.ap);
        }
        return *this;
}
```

However often we can combine these to get something more efficient. In the present case, we `delete` an `A` object and then allocating a new one to receive the copy of the other `A` object. A more efficient strategy is to simply reuse the object instead of destroying it:

```
*ap = *other.ap;
```

This uses the automatically provided assignment operator of the `A` class, which simply copies across all fields. Again, this fine for the `A` class, but not `B`.

This yields the following version of the assignment operator for `B`:

```
const B & operator=(const B &other) {
        if (&other != this) {
                *ap = *other.ap;
        }
```

```
        return *this;
    }
```

2. The compiler will not automatically generate a default constructor, as the programmer has supplied a constructor. It will generate a copy constructor, which initializes the fields memberwise:

```
Person(const Person &other) :
        name(other.name), age(other.age) {}
```

   It will also generate an assignment operator, which assigns the fields memberwise:

```
Person &operator=(const Person &other) {
        if (&other != this) {
                name = other.name;
                age = other.age;
        }
        return *this;
}
```

   It will also generate an empty destructor:

```
~Person() {}
```

   The copy constructor and assignment operator do exactly what we want. The empty destructor is almost right, since the `string` field is a subobject and thus destroyed automatically. The only problem is if the `Person` class is used as a base class. With the automatically generated destructor, the destructors of these derived classes won't be called if their objects are deleted through `Person` pointers. To deal with this this, we need to supply an explicit `virtual` destructor, even though we have nothing to destroy:

```
virtual ~Person() {}
```

3. (a) For variables of primitive type, there is no difference: both initialization and assignment amount to copying the value.

   (b) Objects are initialized by constructors. Initialization from another object of the same type invokes the copy constructor. Assignment is done using the assignment operator. When the object holds resources (e.g. storage) that must be freed by a destructor, these are usually defined to do different things: the copy constructor operates on an uninitialized object, while assignment is called for an object that already has these resources, which must often be freed first.

   (c) Constants can (indeed must) be initialized, but cannot be assigned to.

(d) Initialization of a reference makes it another name for an existing location. Subsequent assignment to the reference assigns to that location. For example, call by reference initializes a local reference parameter by making it an alias for the variable passed as an argument. Assignment to the reference parameter then performs assignment to that argument variable. It is also possible to have reference variables, and the same principle applies. The following example is taken from section 5.5 of Stroustrup's book:

```
int i = 1;
int &r = i;      // r and i now refer to the same int
int x = r;       // x = 1;
r = 2;           // i = 2;
```

4. The idea here is that we will be storing a string in an array, but it may happen that the array is longer than wee need for the number of characters in the string. However it cannot be shorter.

   So now we have two lengths: the length of the string and the length of the array, and thus two integer fields. We need to be very clear about the role of each, so we add a one-sentence comment describing the role of each (always a good idea with every field of an object) and an invariant stating a relationship we will maintain:

```
// length of the string
int len;

// length of the chars array
int array_len;

// Invariant: len <= array_len
```

   When we first construct a string, these two lengths will be the same, namely the length of the C-string we are given to store.

```
public:
        String(const char *s) : len(strlen(s)), array_len(len),
                        chars(new char[array_len]) {
                for (int i = 0; i < len; i++)
                        chars[i] = s[i];
        }
```

   Even though the two fields are the same at this point, we will always use the field `array_len` when allocating the array. Note that `array_len` is initialized from `len`, and `chars` is initialized using `array_len`, so it is essential that `len` is initialized first, followed by `array_len` and then `chars`. The order in which they are initialized is

determined not by the order of the initializers in the constructor, but by the order in which they are declared, so we have carefully declared them in that order.

The default constructor is similar; here both lengths are zero:

```
// default constructor
String() : len(0), array_len(0), chars(new char[array_len]) {}
```

As before, we need a destructor to dispose of the dynamically allocated array:

```
// destructor
virtual ~String() { delete[] chars; }
```

Because the array way allocated with `new[...]`, it must be destroyed with `delete[]`, because the system cannot distinguish a pointer to one element from a pointer to an array.

With the copy constructor, the number of characters we have to store is the length of the other string (not the length of the array in which it is stored). We can store this in an array of that length, or any larger length. We choose to use the string length for the array length:

```
// copy constructor
String(const String &s) : len(s.len), array_len(len),
                chars(new char[array_len]) {
        for (int i = 0; i < len; i++)
                chars[i] = s.chars[i];
}
```

We could have initialized `array_len` as `s.array_len`, or indeed anything greater than or equal to `s.len`.

So far the array length is always the same as the string length. The difference comes with assignment. Previously we always `delete`d the array we had and allocated and allocated a new one of the appropriate size. But if the string being assigned will fit in the array we currently have, we can re-use our old array, but now it might not be full, so we will need our two different lengths.

The outline of the assignment operator always looks like this:

```
// assignment operator
String & operator= (const String &s) {
        if (&s != this) {   // don't copy onto self
                // manage space and copy
        }
        return *this;
}
```

If the string being assigned is the same as the current object (i.e. it is being assigned to itself) then we need do nothing. In any case, we always return the object assigned to by reference, so one can write a chain of assignments like

```
s1 = s2 = s3;
```

Now what goes in the middle bit? We want to copy the other string into an array large enough to hold it. In general, it should have the effect of the destructor followed by the copy constructor. But if the other string fits in the array we have, we can do better.

Note that there are four lengths involved here, two from each object; we must be very careful about which is appropriate at each point.

Firstly, the length of the new string will be the length of the string being assigned. The length of the string that is being overwritten is irrelevant, and can be forgotten:

```
len = s.len;
```

Now before copying the string, we need to ensure we have room to hold it, i.e. our current array is at least as long as the new string. If this is not the case, then we will need to discard our array and allocate a new one of sufficient size, as before:

```
if (array_len < len) {
        delete[] chars;
        array_len = len;
        chars = new char[array_len];
}
```

In that case, we set `array_len` to the new length before allocating the array. Again, we could have set `array_len` to anything greater than or equal to `len` (for example `s.array_len`) but we have opted for the smallest possible value.

Now we have an array of sufficient size to hold the string, either because it was already big enough, or because we have just allocated a sufficiently large array. So we can copy the other string into this array.

```
for (int i = 0; i < len; i++)
        chars[i] = s.chars[i];
```

And we are finished. So here is the complete code for the assignment operator:

```
// assignment operator
String & operator= (const String &s) {
        if (&s != this) {   // don't copy onto self
```

```
                len = s.len;
                // reuse the old array if possible
                if (array_len < len) {
                        delete[] chars;
                        array_len = len;
                        chars = new char[array_len];
                }
                for (int i = 0; i < len; i++)
                        chars[i] = s.chars[i];
        }
        return *this;
}
```

Some people divided the whole code between two cases. That's fine as a starting point, but always re-examine you code to see if it can be simplified. In this case, if the same code occurs at the end of both cases, it can be moved out of the conditional. Similarly if the same code occurs at the start of both cases. In this case you can move all the code out of one of the cases, so the `if` may be simplified.

5. In the statement

```
s1 = String("wheel");
```

a temporary `String` object is created, and initialized from the character string `"wheel"`, using the constructor

```
String(const char *s)
```

This object is then assigned to the variable `s1`, which invokes the assignment operator. This `delete`s the `chars` array in `s1`, allocates a new one of length 5 and copies the characters into it. The temporary object is then destroyed, which invokes its destructor, which `delete`s the `chars` array in it.