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

Solutions to Exercise Sheet 5

1. The function is parameterized by the type of the items in the list:

```
template <typename Item>
void print_list(list<Item> & l) {
```

We will iterate over the elements of this list, so it is convenient to introduce an abbreviation for the type of iterators over these lists:

```
        typedef list<Item>::iterator Iter;
```

Now we use the abbreviation `Iter` we've just defined as the type of the iterator in our loop. The loop has the standard form, with `begin()`, `end()`, `++p` and `*p`:

```
        for (Iter p = l.begin(); p != l.end(); ++p)
                cout << *p << ' ';
        cout << '\n';
}
```

Note the use of `++p` rather than `p++`. (The two have the same meaning, as long as you don't use the value, and for primitive types it doesn't matter which you use. But for iterators, `++p` is much cheaper.)

We can generalize further: the above code works for any sequence container (e.g. `vector` or `deque`), not just `list`. So we can parameterize by the sequence type:

```
template <typename Sequence>
void print_all(Sequence & s) {
        typedef typename Sequence::iterator Iter;
        for (Iter p = s.begin(); p != s.end(); ++p)
                cout << *p << ' ';
        cout << '\n';
}
```

Note that we had to add the keyword `typename` to tell the compiler that `Sequence` is a type.

2. This is very similar to the last question, except that this time we have a `list` of `int`, rather than a parameter type:

```
void double_list(list<int> & l) {
        typedef list<int>::iterator Iter;
        for (Iter p = l.begin(); p != l.end(); ++p)
                *p = *p * 2;
}
```

   Alternatively, the last line could be written

```
                *p *= 2;
```

   Note that there are different uses of `*` here. The first is a unary `*`, which dereferences the iterator, referring to the current element of the underlying container. The second is binary `*`, which here is ordinary multiplication.

3. The `double_list` function generalizes in a similar way to `print_list` above:

```
template <typename Container>
void double_all(Container & c) {
        typedef typename Container::iterator Iter;
        for (Iter p = c.begin(); p != c.end(); ++p)
                *p *= 2;
}
```

4. We begin with an empty vector:

```
    vector<int> v;
```

   and then use an iterator to loop over the list in the usual way:

```
    typedef list<int>::iterator Iter;
    for (Iter p = l.begin(); p != l.end(); ++p)
        v.push_back(*p);
```

5. First, we build the map from words to counts as before:

```
int main() {
        map<string, int> word_count;
        string s;
        while (cin >> s)
                word_count[s]++;
```

The difference is that we no longer need an extra container to list all the words we've seen. This is the same as the set of keys in the map, so we can iterate over the map itself. First we define an abbreviation for the type of this iterator:

```
        typedef map<string, int>::iterator Iter;
```

An iterator `p` over an associative container like a `map` differs from one over a sequence in that `*p` is a pair consisting of a key and an associated value. In our case, the type of `*p` is `pair<string, int>`, an object with fields `first` (the word, of type `string`) and `second` (the count, of type int). As we iterate over all the word-count associations in the `map`, we can refer to each work as `(*p).first` or `p->first` and similarly for the counts:

```
        for (Iter p = word_count.begin(); p != word_count.end(); p++)
                cout << p->first << ": " << p->second << '\n';
```

Then we finish the `main` function in the normal way:

```
        return 0;
}
```