IB9N7 C++ for Quantitative Finance

Lecture 9: Function interfaces

J.F.Reizenstein@warwick.ac.uk

07 December 2015
(Week 10)

Outline

In this lecture, we'll review the following topics:

- Temporaries
- Passing by reference vs. passing by value, now for objects
- Returning by reference vs. returning by value
- Member functions vs. free functions

Objects, references and temporaries

Creating objects in expressions

Last lecture (lecture 8), you saw the syntax

```
Date test2 = Date(9, 12, 2014); // assignment-like syntax
```

Using a similar syntax to the RHS, you can also create object instances in an expression without actually declaring a variable:

```
Date(9, 12, 2014).output_shortform();
```

This then creates a **temporary** variable (though the variable in this case does not have a name visible in its scope).

This can be useful for returning an object by value from a function (see slide 9).

References

There are multiple interfaces that you could provide to solve the problem of calculating the date after a given date (can't use all at once!):

```
// free functions
namespace DateStuff
{
    Date next_date(Date start);
    Date next_date(const Date & start);
    void advance_date(Date & date);
    Date & advance_date(Date & date);
}

// member functions
Date Date::next_date() const;
void Date::advance_date();
Date & Date::advance_date();
```

Actually, you could also put the `Date` class within the `DateStuff` namespace.

Passing by value doesn't change original

Suppose that `next_date` was defined as follows:

```
Date next_date(Date start)
{
    // ... calculate day, month, year ...
    start.set_date(day, month, year); // only modifies a copy
    return start;
}
```

Then the following would result in the output as in the comments:

```
Date test(9, 12, 2014);
test.output_shortform(); // 09/12/2014
Date x = DateStuff::next_date(test);
test.output_shortform(); // still 09/12/2014
x.output_shortform(); // 10/12/2014
```


Free or members?

Note that the `next_date` or `advance_date` functions don't need to know anything about how the `Date` class implementation in order to work.

However, such information might allow those functions to be implemented more efficiently.

If the class implementation were to change, then the member function forms (which may rely on implementation details) would potentially get broken, whereas the free functions know nothing of the implementation, and therefore provide greater encapsulation.

Free functions also mean that the class definition can be simpler.

Scott Meyers therefore argues that free functions are preferred unless you need to worry about template utility.

Temporary objects

C++ implicitly creates/copies/destroys objects.

Can be unexpected and often unwanted.

Temporary objects created when:

- Passing objects by value to a function.
- Returning objects by value.

Unnecessary temporaries can waste time and memory.

Write code to avoid large temporary objects if possible, e.g:

Pass large objects by `const &`, not by value.

Note that returning temporaries or locals by reference is not permitted, and returning them by const reference leads to undefined behaviour.

Date next_date(Date start)

```
1  Date next_date(Date start)
2  {
3      int day = start.get_day();
4      int month = start.get_month();
5      int year = start.get_year();
6
7      // ... calculate day, month, year ...
8
9      return Date(day, month, year);
10 }
```


Date next_date(const Date & start)

```
1  Date next_date(const Date & start)
2  {
3      // Exactly the same as above, but passing around
4      // a reference reduces the overhead of an unnecessary
5      // temporary.
6  }
```


void advance_date(Date & date)

```
1  void advance_date(Date & date);
2  {
3      int day = start.get_day();
4      int month = start.get_month();
5      int year = start.get_year();
6
7      // ... calculate day, month, year ...
8
9      date.set_date(day, month, year);
10 }
```


Date & advance_date(Date & date)

With a function of this kind, often helps to return a reference to the parameter being modified, to allow chaining of function calls.
e.g. `advance_date(test).output_shortform();`,
instead of
`advance_date(test); test.output_shortform();`

```
1  Date & advance_date(Date & date)
2  {
3      // as before
4      date.set_date(day, month, year);
5
6      return date;
7  }
```

Member function versions

The member function versions are identical, but instead of taking a `Date` parameter, they operate with reference to `*this`.

(The asterisk is necessary because the `this` keyword gives a pointer instead of a copy.)

Lab objectives

- Keep working on understanding OOP.
- Understand the new lecture content of today. Check that the interfaces of your `next_date` and `advance_date` functions were similar to those shown.
- Get your coding style etc checked.
- Prepare for the class test!