IB9N7 C++ for Quantitative Finance

Lecture 8: Introduction to OOP in C++

J.F.Reizenstein@warwick.ac.uk

30 November 2015
(Week 9)

Classes

Simple class

- In C++ it is very easy to create a new type (a structure) which is made from a compound of other types. If a type is thought of as a mathematical set, this is like creating a cartesian product of sets.
- This facility to create our own types underlines Object-oriented programming which is discussed in this lecture. The next slide is just an example of its use alone.

Simple class

```
#include<string>
#include<vector>
#include<iostream>

class Name{
public:
    std::string firstName;
    std::string lastName;
};

int main(){
    Name me;
    me.firstName = "Jeremy";
    me.lastName = "Reizenstein";
    std::vector<Name> names;
    names.push_back(me);
    for(Name& n : names){
        std::cout<<n.firstName<<" "<<n.lastName<<"\n";
    }
}
```

Procedural programming

So far, we've seen only the procedural programming approach, where:

- Set of instructions, executed in order.
- Functions call functions.

May have restrictions on visibility, but

- ! Every function may be visible to every other.
- ! All data may be visible everywhere, or painstaking to pass around.
- ! Validation of data needs to be repeated.
- ! Difficult to see how the program works.

Temptation:

- ! Call anything from anywhere.
- ! Modify any data you like.

OOP

Object-oriented programming

Object-oriented programming (OOP) concepts help to reduce risk, cost and coupling when programming.

Thus far, most of the things we've seen aren't too different from the standard things one finds in C, the language upon which C++ is based.

Object-oriented programming is a major concept which makes C++ different to C.

There are other OOP languages, but some do not support OOP fully (e.g. VBA).

- C++ does have full support.

Classes and objects

Definition: Classes

A C++ class is a programming construct which can hold data and provide (and restrict access to) a set of facilities for manipulating those data.

Definition: Objects

An object is an **instance** of a class which has real memory allocated to it.

- A **class** (or the non-`static` part thereof) is a blue-print for creating **objects** that should all behave in the same way.
- Classes are smart user-defined data types, and objects are actual variables of those types.

Benefits of OOP

Classes:

- Can **encapsulate** data:
  - describe what data they will store
  - allow (only) permitted operations on those data
  - and any necessary housekeeping (to allow objects to be created, copied, assigned, destroyed correctly).
- Provide **interface** functions to access and manipulate the data. Abstracts away the implementation.
- Allow for **safer** programming.
- Enable **efficient** code-reuse.
- **Easier** to test, understand etc.
- Can **reduce** coupling (both code and compile coupling).

OOP concepts

Abstraction:	Worry only about the interface, not the implementation. All data types provide data abstractions.
Encapsulation:	Store data and functions acting on those data in the same place. Access to the data is controlled as part of the interface.
Inheritance :	Different classes that are functionally related and need to do similar things can share code.
Polymorphism:	"Many forms." Can treat different objects in the same way if they share a common interface. Can be more dynamic than just function overloading.

Program design in an OOP world

Split code up into classes and objects.

Each class should generally capture exactly one concept (conceptually self-contained).

Provide the **necessary** member functions for each class (don't implement every possible thing you can do with an object).

Carefully design beforehand: classes, objects and their functions and interactions.

1. Specify the types.
2. Declare objects of each type.
3. Set up interactions between objects. (Objects send messages and requests to one another, i.e. act on one another, via their interfaces.)

OOP consequences

Objects become self-contained units,

program is assemblage of objects, easier to understand.

Main program:

Creates a set of objects.

Asks those objects to do things.

Different way of thinking to procedural design.

At the same time, don't want classes to grow too large to do every possible task imaginable.

Some functionality can be provided in an appropriate namespace, rather than a class.

Interface and implementation

Similarly to functions, classes are defined by both

Their interface:
: Signatures of the **public members** that can be accessed by external objects, giving the object its external functionality.

  Also, any functions outside the context of the class that nevertheless relate to that class.

Their implementation:
: The bodies of the public member functions, and private data and functions used by the interface functions.

Unfortunately, in C++, the interface and implementation cannot be specified entirely separately.

Private functions and data have to be specified in a class definition together with public member functions.

OOP in C++

This lecture attempts to give an overview of some of the main aspects of OOP in C++.

Unfortunately, there are a lot of aspects that need to be covered simultaneously in order to provide a reasonable level of understanding.

We will revisit many of these aspects in more detail at a later stage.

Defining a class

A class definition describes what members are contained in the class.

Class definitions

```
class ‹identifier›
{
    ⋮
};
```

- The `class` keyword indicates that we are declaring or defining a class.
- The members of the class are defined within the curly-braces.
- **Remember:** Unlike function definitions, need to end class definitions with a semi-colon!

A class definition example

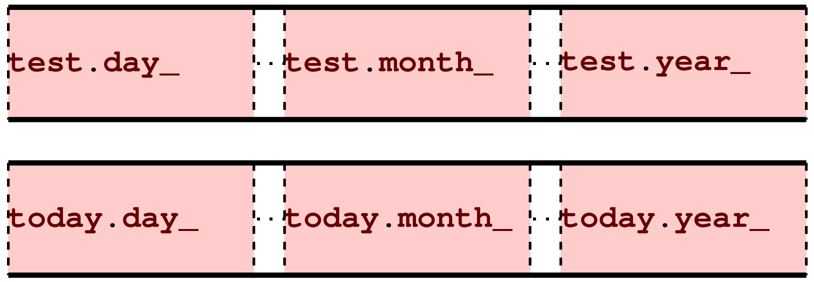
Example: (bare) Date class

```cpp
// place within Date.hpp (with the usual header guards)
class Date
{
  public:
    // public static members
    static bool is_valid_date(int day, int month, int year); // validation
    static bool is_leap_year(int year); // helper function
    static int month_length(int month, int year); // helper function

    // public non−static members
    Date(int day, int month, int year); // constructor
    void output_shortform() const; // print to std::cout

  private:
    // private static members
    static void check_valid_date(int day, int month, int year);

    // private non−static data
    int day_;
    int month_;
    int year_;
};
```

Memory: diagram

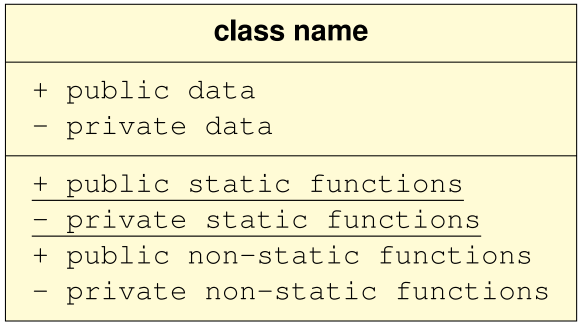
Suppose we have two variables (object instances) of this class data type `Date`, called `test` and `today` say.

Data for the same object instance are stored together in memory, but we can't say anything about ordering or contiguity etc.



UML diagram ☆

Sometimes it helps to visualise a class using a class diagram.

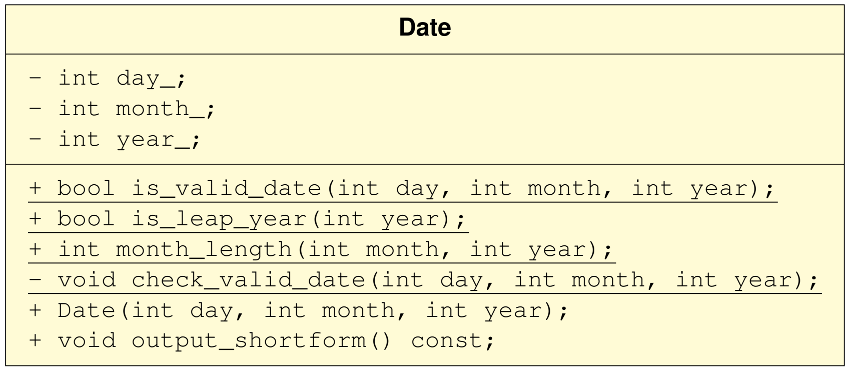


class name

```
+ public data
- private data

+ public static functions
- private static functions
+ public non-static functions
- private non-static functions
```

UML diagram for `Date` ☆

Slightly abused from the UML standard (unrelated to C++)!



Date

```
- int day_;
- int month_;
- int year_;

+ bool is_valid_date(int day, int month, int year);
+ bool is_leap_year(int year);
+ int month_length(int month, int year);
- void check_valid_date(int day, int month, int year);
+ Date(int day, int month, int year);
+ void output_shortform() const;
```

Class interface

Only the `public` entities (and external relevant entities) form part of the interface.

How the class behaves internally (in the example, the functionality provided by `check_valid_date` and the storage of `day_`, `month_` and `year_`) is not part of the interface.

Could internally store the number of seconds since some epoch (01/01/1970 say) instead of (or in addition to) day, month, year.

- Interface would be the same.
- Client need not know.
- If storing both, could keep the two representations in sync due to the controlled access.

Encapsulation

Data (state) related to a class should generally be internal to the class. Specifically it should be:

1. part of the class definition; and
2. `private`.

Each object instance of the class gets its own copy of the non-static data, with no relation to the non-static data of other object instances.

- Functions that can act directly on the data are part of the class.
- Data only accessible to external clients through a controlled interface.
- Essential to maintain **state invariants**.
- Objects should always be in a valid state.
- A class has sole responsibility for the state of its objects.

Access specifiers

Encapsulation of data is achieved through access specifiers which specify whether members within the class are accessible from outside.

Entities within a class can be:

- `public`: member is accessible from outside the class.
- `private`: member is only accessible by functions within the class.
- `protected`: member is accessible to functions within the class and functions in derived classes (see inheritance in a later lecture).

Can also declare friends, which confuses this a little.

Default access specifiers ☆

Default access

By default, members are `private`, but it is good practice to explicitly mark such members as `private`, in case there are (ever) any members with other access levels.

Note to C-programmers:

`struct`s in C++ are `class`es with `public` default access.

In C++, `struct`s can contain member functions etc.

Generally prefer to use `class` instead of `struct` in C++.

Member functions

Member functions

A member function is a function which is declared to be part of a class.

It has access to all public and private data and members of the class.

- A member function is declared (and possibly defined — see later) within the class definition.
- Note: a function that is not a member function of any class is called a **free function**.

Non-static member functions

Non-static member functions are invoked in the context of a particular *object*, rather than in the context of the class itself.

From clients, this is done through the component selection operator `.` on an object instance.
e.g. if `test` is a `Date` object, then can call
`test.output_shortform();`

Non-static member functions have a hidden parameter corresponding to the memory address of the object instance in which they were invoked.

`this` keyword ☆

The keyword `this` gives a ‘ pointer ’ to the current object. In practice, we use `*this` to get a reference.

Member variables

Member variables

A member variable is a variable which is declared to be part of a class.

In instances of the class (objects), the non-static member variables hold the data associated to the instance in question.

It is good practice to follow a naming convention for member variables, eg to postfix them with an underscore (_). I will mostly use this convention. Another common one is to prefix them with `m_`.

While member variables can be `public`, as mentioned earlier, it is best practice to always make them `private`.

Getters and setters

Provide “getters” and/or “setters” to access the private data from outside the class, if it is desired to allow such access.

Generally don’t provide direct access to the data (even if usage need not be restricted currently).

- One day may need to control/log access to the data, and would then otherwise need to change the interface.
- Little to no overhead of going through a member function.

If the setter fails (data does not validate) then the class should be left in the same state as it was prior to invoking the setter (known as the **strong exception guarantee**).

No way to access or mutate Date yet

In the previous example, the date cannot be read or modified after the object has been created, since there are (respectively) no accessor or mutator functions in the class (only an output function, which doesn’t provide access in the same way).

e.g. The `day_` variable is `private` and cannot be accessed by any function apart from those defined within the class.

Our `Date` class would be much more useful if we also provided such accessor and mutator functions as part of the public interface.

We will do this on the next slide.

Date accessor/mutator functions

In the `public` section of the class definition (or in a new `public` section), we can add the following prototypes:

```
int get_day() const; // accessor (getter)
int get_month() const; // accessor (getter)
int get_year() const; // accessor (getter)

void set_date(int day, int month, int year); // mutator (
    setter)
void set_day(int day); // mutator (setter)
void set_month(int month); // mutator (setter)
void set_year(int year); // mutator (setter)
```

We could (but won't yet) also declare a function `get_number_of_seconds_since_epoch` to access a different representation of the state of the class.

`const` functions

Can declare non-static member functions to be `const`,

as part of the signature following the closing parenthesis of the parameter list.

Doing so prevents them from modifying data members of the current object instance directly,

or indirectly, by preventing them from calling other non-`const` member functions that may so modify the data members.

Getters should be `const` (as on the previous slide), as well as other non-mutator functions (like `output_shortform`).

Best to get the `const` correctness right first time around.

Benefits of encapsulation

- The previous example encapsulated the `day_`, `month_` and `year_` variables.
- Member functions must be used to access and change the state.
- Encapsulating the private data this way allows the class to *validate* the input, and thus ensures *data integrity*.

Class design

As a general rule, make all data members `private` and create *getters* and/or *setters* to access them as necessary.

Clients

The code using a class is called a **client** of the class.

The client:

need not know what data it has internally,

and **should not** know.

Encapsulation helps provide this separation of knowledge.

E.g. Client doesn't really know that `day_` member even exists.

If `test` were a `Date` object, referring to `test.day_` from client code would be an error.

Refer to `test.get_day` or `test.set_day` instead.

Static members

`static` in the context of a class member means that it is shared across all instances of the class.

In fact, static members still exist and can be accessed even if no class instance (object) exists at all.

For `static` members, behaves as though the class were a namespace, rather than an object blue-print.

In the `Date` example, we used this to provide the function `is_valid_date` within the `Date` "namespace" (as the function is relevant to dates).

Would not be useful as a non-static function since every `Date` object instance will correspond to a valid date by design!

Could define it as a (non-static) free function, but best to keep together with the class.

Constructors

Constructor

A constructor is a special member function which is called when an object is created.

The creation could be due to being explicitly or implicitly instantiated/assigned/copied, etc.

Its primary (but not necessarily exclusive) role is to initialise any variables defined within the class.

- A constructor always has the same name as the class.
- A constructor has no return type (not even `void`).
- Multiple constructors (overloads with unique signatures) may exist for a class. This is a very useful demonstration of overloading.

Constructor exceptions

If the arguments passed to a constructor are not semantically valid according to the contract of the class

but are syntactically valid so that the compiler will accept attempts to create an object instance with those invalid arguments,

then the constructor needs to signal (**throw**) a run-time error condition that causes the object construction to be abandoned and rolled back.

This run-time error condition mechanism is the exception mechanism and will not be explained today.

For now, it is enough to note that

```
throw std::runtime_error("description")
```

is one way to indicate something happened which should not have happened, and at the moment, will cause the program to exit/crash.

Creating objects

Defining instances of a class is just like defining variables of fundamental types.

The data type identifier is now the class name.

Parameters can be passed to the constructor as follows. As usual for function-like initialisation syntax, if the constructor takes no arguments, the parentheses are omitted.

```
Date test1(15, 12, 2015); // function−like syntax
//  Declares an object variable called test1 of type Date
//  and uses the constructor with the (int, int, int) signature.
Date test2 = Date(15, 12, 2015); // assignment−like syntax
//  Same but for identifier test2.
```

Class implementation

So far, the class is only *defined*, but not implemented. We now need to implement all the member functions.

- The members in the class definition are essentially function prototypes.
- The member function definitions are written in a similar way to normal function definitions, but with a few important differences.

Member function implementations

Member function implementations can be provided in two different ways:

1. The function definitions are usually written outside of (and must come after) the `class` definition.
   To indicate that a function definition belongs to a particular class, it is preceded by the class name and the scope operator.
   The keyword `static` is only used for member declarations within the class definition.
2. A function definition can also be written *within* the `class` definition **instead** of providing a prototype.
   Implicitly declares the functions `inline` to allow them to be defined inside a header file without potentially causing multiple definitions.

Normally best to use the standard separation of header and implementation files in conjunction with method 1.

Possible implementation: setters

```cpp
void Date::set_date(int day, int month, int year)
{
    check_valid_date(day, month, year);
    day_ = day;
    month_ = month;
    year_ = year;
}

void Date::set_day(int day)
{
    check_valid_date(day, month_, year_);
    day_ = day;
}

// ...
```

Possible implementation: getters

```cpp
int Date::get_day() const
{
    return day_;
}

int Date::get_month() const
{
    return month_;
}

int Date::get_year() const
{
    return year_;
}
```

Initialisation lists

Initialisation lists are a special feature that can be used (only) in constructors.

They provide a way of directly setting the initial values for non-static member variables in an object.

It is generally better to initialise variables than for them to be created uninitialised and then assigned to. For objects, initialisation happens in the initialisation list.

The initial values can be set to any expression, not just parameter identifiers.

Initialisation list syntax

- The **initialisation list** is part of the implementation of the constructor, goes between the parameter list and the opening brace, and is started with a colon.
- Initial values are placed in parentheses following the name of a member variable.
- Successive variables are delimited by commas.

Easier to see by example:

Example: constructor with an initialisation list

```
Date::Date(int day, int month, int year)
    : day_(day), month_(month), year_(year)
{
    check_valid_date(day, month, year);
}
```

Initialisation lists explained

The following example, which you might use if you did not know about initialisation lists, creates the object with the member variables uninitialised, and then assigns to them:

Example: without an initialisation list

```
Date::Date(int day, int month, int year)
{
    day_ = day;  // should use initialisation list instead
    month_ = month;  // should use initialisation list instead
    year_ = year;  // should use initialisation list instead
    check_valid_date(day, month, year);
}
```

For this simple example, the effects of assignment are essentially the same as those of initialisation, but initialisation is still better.

Naming classes

Best practice: begin the names of your own classes with capital letters.

Classes that form part of the C++ standard library (and fundamental types) are defined with initial lower case letters to distinguish them.

When dealing with object instances, again use lower case initial letters to distinguish those from classes.

Compile decoupling

Saving classes:

- Put the definitions of the non-inline class member functions into a separately compilable file, named ‹ClassName›.cpp.
- Put the class definition (and inline member function definitions) into a (guarded) header file, named ‹ClassName›.hpp.
- Make sure to use the correct case of filename to match the class name.
- `#include` the header file into the implementation file. If the full class definition is not required, minimise compile coupling by using a forward declaration of the form
  `class ‹ClassName›;`
  instead of the `#include` directive.

Example: using the Date class

```cpp
#include <iostream>
#include <cstdlib>
#include "Date.hpp"


void date_test_main()
{
    Date test(15, 12, 2015);
    test.output_shortform();
    Date(9, 12, 2014).output_shortform();
    Date today(30, 11, 2015);

    std::cout << "The␣year␣of␣the␣test␣is␣" << test.get_year() << std::endl;

    //std::cout << test.day_ << std::endl;
    //std::cout << Date::get_day() << std::endl;
    //Date::output_shortform();

    //std::cout << (Date(35, 12, 2014)).is_valid_date(1, 1, 2014) << std::endl;

    test.set_year(2015); // you wish
    test.output_shortform();

    // Uncomment this for a crash. We will see how to handle this better soon.
    //Date baddate(31, 11, 2014);

}

int main()
{
    date_test_main();
    return EXIT_SUCCESS;
}
```

Lab objectives

- Understand the program that demonstrates the `Date` class!
- Add/think about appropriate additional functionality for that class.
- Create a `Point` class that has appropriate member variables and functions with the appropriate access restrictions and validation.