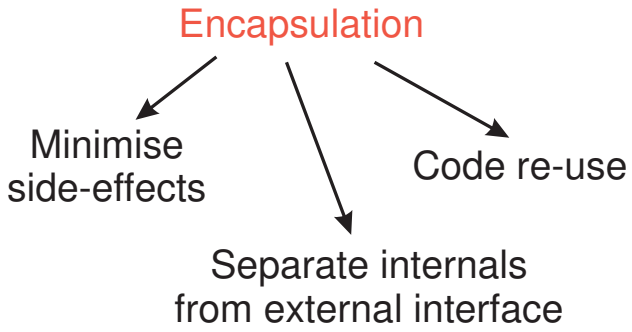
Basic C++ concepts

This lecture will introduce you to some fundamental ideas arising from C++.

- Classes and objects
- Creating and destroying objects
- Data members
- Member functions
- Access control of class members

But first, why is object oriented design thought to be such a good idea ?



What is a class ?

A class is an extension of a C-style structure ...

A class is a powerful ADT, which includes not only data members, but also *member functions*.

(a member function is a function listed as belonging to a particular class - the intention is that the member functions provide the behaviours necessary to drive the class)

A class controls access to all its members; we can very precisely specify which parts of a class are accessible from outside, and which are strictly internal. Hiding both data and functionality is a very powerful idea.

What is a class ?

Well defined mechanisms exist for the inheritance of certain properties from a *base* class to a *derived* class.

(this represents an extremely effective code re-use mechanism. If an existing class provides certain core functionality, we can use it by deriving a new class from it.)

So, what is an object ?

An *instantiation* of a class.

Consider: `double eigenvalue;`

`Eigenvalue` is an instantiation of a `double`.

`double` is a type; it does not use storage but `eigenvalue` is an instantiation; it does.

So it is with objects. A class is a super-type, and an object is an entity which requires memory, and is of a certain class.

Anatomy of a class

What can exist inside a class ?

Data members - these are like the elements inside a C-style struct.

Member functions - this is the most important idea in C++...

A member function is a function which is defined in the context of a particular class. In other words, a class contains data and functionality.

Member functions are intended to be the primary mechanism for the manipulation of objects of a given class.

Can I see one ?

Sure...

```
class complex
  {
  double real;
  double imaginary;
  };
```

This is a simple class declaration, and it allows us to go on to write things like :

```
complex input_impedance;
```

This isn't very exciting, because we have no member functions, and no access control. We could have done this in C...

```
typedef struct
  {
  double real;
  double imaginary;
  } complex;
```


The differences between C and C++ start to become apparent when we want to think about providing functions to manipulate our complex numbers...

In C, we write functions which take the complex type as parameters, and generate complex results. This is fine, but...

- How do we keep these support functions together ?

- How do we prevent name conflicts between our complex functions, and other parts of the software ?

- how do we support nice syntax; `add_complex(Z1,Z2)` is clumsy.

- How do we hide the functionality of the internal methods ?

Answer; we don't.

Name space conflicts can be quite serious in large software projects; if the language doesn't support C++ style encapsulation, we are forced to use awkward variable name prefixes...

C++ philosophy

We split the problem into sets of classes. The resulting class hierarchy attempts to express the dependencies and inter-relationships between the elements of the solution.

This process is non-trivial. Long after the mechanics of C++ have been absorbed, intelligent class hierarchy design will still be a challenge! After the classes have been designed, it simply remains to "fill in" the methods...

The rise and fall of objects.

Unlike a simple scalar variable (say an integer loop counter), objects make a fuss about coming into the world, and leaving it.

Working under the assumption that an object is a non-trivial entity (often all too true...), the language lets us make implicit calls to functions to handle the initialisation and clean up. These calls are to the classes' *constructors* and *destructors*.

Whenever an object is created, the system automatically allocates enough memory to hold all its data members, then calls a constructor, if one is provided. The constructor function is called in the *context* of the fledgling object.

Constructors

This means that the object for which the constructor is working is implicit.

A constructor function is identified as a member function which has the same name as the class.

Let's add a constructor to our complex class...

```
class complex
  {
  private:
    double real;
    double imaginary;
  public:
    complex();
  };
```

Note the access control; the constructor should be available to everyone.

Constructors

C++ allows multiple constructors; this allows us to use the class to initialise each object in different ways.

Consider :

```
complex::complex(double re,
                 double im)
{
real = re;
imaginary = im;
}
```

C++ selects the constructor to be used at compile time, on the strength of the signatures of the constructors and the call given. C++ has complex type resolution rules, which allow it to both promote and demote types to find a constructor which fits.

Constructors

Constructors can be quite bizarre...

```
complex::complex(char initial)
{
if (initial == 'j')
   {
   real = 0;
   imaginary = -1;
   }
else
   {
   real = imaginary = 0;
   }
}
```

Our simple class does not need special clean up code, but if it had allocated space or opened a file during each object's construction, we would need another call to kill the object nicely...

Destructors

Destructors are the Grim Reapers of object world. They are only called when an object is to be expired.

They are identified by using the same name as the class, but with a "~" prefix:

```
complex::~complex()
{
cout << "One less complex..."
     << endl;
}
```

As with construction, the memory used by the data members of the object are automatically handled by the system.

Use destructors to clean up additional resources used by the object, or to keep track of how many objects of a certain class still exist.

Member functions

Member functions are those functions belonging to the class. They must be listed in the class declaration, and their methods are identified like this :

```
complex::conjugate()
{
imaginary = -imaginary;
}
```

It is not unusual (or necessarily bad) for classes to have many data members and member functions. However, some problems are amenable to neat hierarchies, and then we need to worry about access control - which members (functions or data) are visible to which other parts of the software. This is critical, because it directly affects data hiding.

Access control

Three levels :

public:

Accessible from everywhere.

private: (default)

Accessible only by member functions.

protected:

A hybrid. Treated as private, except also visible to friend functions, and those from derived classes.

A friend function is a function listed in the class declaration as a friend. In C++ land (at least), friends give access to each other's private parts...

I'll leave derived classes for later.