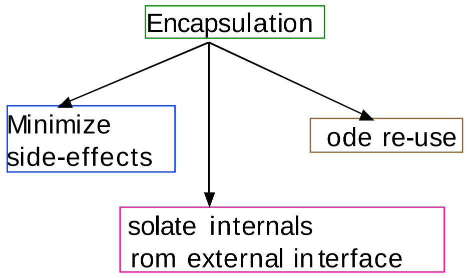
What are the Basic C++ concepts

The following basic ideas are contained in C++

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

Why is object-oriented design thought to be a good idea?



What is a class?

A class extends the notion of a structure in C

A class is a powerful ADT, including not only data members, but also member functions (operators).

Each member function belongs to a particular class - the idea is that these member functions is that they provide the only means to access the data items.

A class ***controls*** access to its data members through its member components; thus we can precisely specify which parts of a class can be accessed from the outside, and which parts are strictly internal (***private***). The idea of hiding both data and controlling (limiting) access is very powerful.

To improve upon this idea we also have a well-defined mechanism for ***inheriting*** certain properties from a base class to a ***derived*** class.

From this we receive an extremely effective code re-use mechanism. If an existing class provides certain basis capabilities, we can incorporate them by deriving a new class from it.

What is an object?

An object is an instance of a class.

```
double eigenvalue;
```

In this definition, eigenvalue is an instance of an uninitialized double.

double is a type, it does not require memory, but eigenvalue is an instantiation--it needs 8-bytes.

It is the same with objects. A class is a ***supertype***, and an object is an entity which requires memory, and is of a certain class (in our example type).

Anatomy of a class

What can exist inside a class?

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

Member functions--these are the operators in an ADT, and form an important contribution of C++.

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

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

An example

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

This a simple class declaration and it allows us to create an instance with:

```
Complex variable;
```

Note: we could also have done this as easily in C with:

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

The differences between C and C++ only become apparent when we think about how to provide access functions to manipulate our complex numbers.

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

* How do we keep these support functions together?

* How do we prevent name conflicts between our local complex functions and other software?

- How do we provide a nice syntax? Given:

```
  Complex Z, Z1, Z2;
          Z = add_complex (Z1, Z2)
```
is clumsy-looking C-code.

Better is:
```
          Z = Z1 + Z2;
```
this is possible in C++, provided the appropriate operators **=** and **+** are created (using operator overloading concept)

* How do we hide the capability of the internal methods?

The short answer is that in C we can't.

Name conflicts can be quite serious in large software projects. If the language does not support C++ style encapsulation, we are forced to use awkward variable names and consistent prefixes...

C++ philosophy

Split the problem into sets of classes. The resulting hierarchy attempts to express the dependencies and relationships between the components of the solution.

The process is non-trivial. Long after you have absorbed the mechanics of C++, the design of a clean and rational class hierarchy will remain a challenge! Once classes have been designed, all that remains is to fill in the methods (access functions).

The rise and fall of an object

Unlike simple scalar variables (like an integer loop counter), objects need to be created and removed explicitly (and with some fuss).

Assume that an object is a non-trivial entity (all too true), C++ lets us make implicit calls to helper functions to handle the initialization and to clean up. These implicit calls are to the *constructors* and *destructors* of the class.

When an object is created, the system automatically allocates enough memory to hold all its data members. It then calls the provided constructor (or a default one if necessary). The constructor function is called in the context (framework, environment) of the emerging object.

Constructors

To reduce the need for more identifiers, the constructor functions have the same name as the class. There is no confusion in the computer's mind.

Consider our earlier complex class...

```
class Complex {

private:
    double real;
    double imaginary;
public:
    Complex( );       // our constructor prototype
}
```

First note that now the data items real and imaginary will be hidden from the outside world (are private), but that the access control mechanism (just the constructor for the moment) will be usable by everybody--is public.

The "body" of the constructor can be specified inside the class, but if it is lengthy then it is usually done outside as follows. Access functions defined inside a class are said to be "inline functions" and in some systems may be more efficient (faster).

```
Complex :: Complex (double re, double im) {
    real = re;
    imaginary = im;
}
Complex Z (4.3, 5.7);        // Z = 4.3 + 5.7j;
```

It is also common, for a class to have more than one constructor to handle default or special cases, but we will deal with that later.

At compile time, C++ selects which constructor to use for any specific instantiation of a class, depending on the specific declaration/definition of the class. C++ has special rules which help it resolve apparent conflict.

It is possible to write quite bizarre or involved constructors, but these are not necessarily recommended.

```
Complex :: Complex (char initial) {
    if (initial == 'j') {     /* constructor one parameter (a char) */
        real = 0;
        imaginary = -1;
    } else {
        real = imaginary = 0;
    }
}
```

This particular simple class does not need a special destructor, since the normal procedure exit frees local variables. But had it used new (or malloc) to dynamically allocate space, then it would need a companion destructor to give the space back when the object is released. Similarly, if the constructor opened a file, the companion destructor would flush and close that file, as well as eliminate all trace of the object nicely.

Destructors

These are the "grim reapers" of the object world. They are only called when an object is being eliminated.

Again they are identified by the same name as the class, but with a **"~"** prefix:

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

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

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

Member functions

Member (access) functions must be listed in the class declaration, and so belong to the class. Their operation can be defined inside the class, or outside as is more common:

```
void Complex :: conjugate( ) {    // being declared outside
    imaginary   = - imaginary;
}
```

It is common (and not necessarily bad) for classes to have many data members and member functions. Here we will focus on a few elegant problems that are amenable to tidy hierarchies. Thus we will have to worry about access control--which members (function 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 to everyone from anywhere. Member functions are put here.

private (default):
    Accessible only by member functions

protected:
    A hybrid. Treated as private, except they are also visible to ***friend*** functions, and those functions from ***derived*** classes.

A friend function is a function listed in the class declaration as a friend. In the C++ world, at least, friends are given access to the private parts...

For the moment we will set-aside derived classes (though they are the heart of code sharing in C++ and so cannot be omitted).