Abstraction

Abstraction

It's rather obvious that if we split any task into a number of smaller tasks which can be completed individually, then the management of the larger task becomes easier. However, we need a formal basis for partitioning our large task into smaller ones. The notion of *abstraction* is extremely useful here. Abstractions are high level views of objects or functions which enable us to forget about the low level details and concentrate on the problem at hand.

To illustrate, a truck manufacturer uses a computer to control the engine operation - adjusting fuel and air flow to match the load. The computer is composed of a number of silicon chips, their interconnections and a program. These details are irrelevant to the manufacturer - the computer is a black box to which a host of sensors (for engines speed, accelerator pedal position, air temperature, etc) are connected. The computer reads these sensors and adjusts the engine controls (air inlet and fuel valves, valve timing, etc) appropriately. Thus the manufacturer has a high level or *abstract* view of the computer. He has specified its behaviour with statements like:

"When the accelerator pedal is 50% depressed, air and fuel valves should be opened until the engine speed reaches 2500rpm".

He doesn't care how the computer calculates the optimum valve settings - for instance it could use either integer or floating point arithmetic - he is only interested in behaviour that matches his specification.

In turn, the manager of a transport company has an even higher level or *more abstract* view of a truck. It's simply a means of transporting goods from point A to point B in the minimum time allowed by the road traffic laws. His specification contains statements like:

"The truck, when laden with 10 tonnes, shall need no more than 20l/100km of fuel when travelling at 110kph."

How this specification is achieved is irrelevant to him: it matters little whether there is a control computer or some mechanical engineer's dream of cams, rods, gears, *etc.*

There are two important forms of abstraction: functional abstraction and structural abstraction. In functional abstraction, we specify a function for a module, *i.e.*

"This module will sort the items in its input stream into ascending order based on an ordering rule for the items and place them on its output stream."

As we will see later, there are many ways to sort items - some more efficient than others. At this level, we are not concerned with how the sort is performed, but simply that the output is sorted according to our ordering rule.

The second type of abstraction - structural abstraction - is better known as object orientation. In this approach, we construct software models of the behaviour of real world items, i.e. our truck manufacturer, in analysing the performance of his vehicle, would employ a software model of the control computer. For him, this model is abstract - it could mimic the behaviour of the real computer by simply providing a behavioural model with program statements like:

Objects and ADTs

In this course, we won't delve into the full theory of object-oriented design. We'll concentrate on the pre-cursor of OO design: abstract data types (ADTs). A theory for the full object oriented approach is readily built on the ideas for abstract data types.

An **abstract data type** is a data structure and a collection of functions or procedures which operate on the data structure.

Abstract data type (ADT) is A data structure and a set of operations which can be performed on it. A class in object-oriented design is an ADT. However, classes have additional properties (inheritance and polymorphism) not normally associated with ADTs.

To align ourselves with OO theory, we'll call the functions and procedures **methods** and the data structure and its methods a **class**, *i.e.* we'll call our ADTs classes. However our classes do not have the full capabilities associated with classes in OO theory. An instance of the class is called an *object* . Objects represent objects in the real world and appear in programs as variables of a type defined by the class. These terms have exactly the same meaning in OO design methodologies, but they have additional properties such as inheritance that we will not discuss here.

Constructors and destructors

The create and destroy methods - often called constructors and destructors - are usually implemented for any abstract data type. Occasionally, the data type's use or semantics are such that there is only ever one object of that type in a program. In that case, it is possible to hide even the object's `handle' from the user. However, even in these cases, constructor and destructor methods are often provided