Modular Programming

Modular programming is a generally considered as good practice in software design and it simply means that program functions are broken down into modules. The term *module* is a generic term for such a unit of code and each one of which accomplishes one function and contains all the source code and variables needed. Each language has its own term or terms for modules. For example C has functions, Java has methods, Pascal has procedures and functions, Fortran has subroutines and functions. Modules need to be called from the program or from other modules and this is carries out through an *interface* which defines the input and output format for the module. The interface is defined as part of the definition of the module and the same interface is used to format each call of the module.

Benefits of modular programming

Modularisation is unnecessary for small programs. However, for larger programs, dividing the the code into modules that perform clearly defined functions can have many advantages. For example

1. Each module has separate responsibilities,
2. Modules can be written by different programmers showing a demarcation of personal responsibility,
3. If more functionality needs to be added to the system then the modules that can take on the responsibility of the new functionality can easily be identified, or new modules can be linked into the system to support the new responsibility,
4. Modules - once written - can be used as building blocks for use in new software products,
5. If there is a 'bug' in the system then the functional unit that includes the bug can more easily be identified.

Format of modular programming

In some languages – like Fortran and Pascal – there are explicitly two types of modules (procedures and functions in Pascal or subroutines and functions in Fortran). Hence there are naturally two forms of module. A function generally has the following interface:

<return type> <function-identifier> (<variables>)

in which the <function-identifier> is the name of the function, the <variables> will have values and they can be considered to be the input to the function and the <return type> states the form of the output. The alternative format is that of a procedure (or subroutine) and its interface has the following format

<procedure-identifier> (<variables>).

In this case there is no formally specified output. However, often some of the variables are given values as input and others are assigned values within the code of the procedure.

Languages that only one type of module explicity have only the function defined above. However, they usually get round the apparent deprivation of the procedure by having a *void* function. A void function has the following form

void <function-identifier> (<variables>)

A void function 'returns' nothing and hence the format is then very similar to that of a procedure.

Object-oriented programming

Over recent decades, particularly with the popularity of the programming languages of Java and C++, the concept of modularisation has been taken to a new level. In object-oriented programming a software system is further separated into a set of *objects*. Every computer program simulates aspects the real world relevant to the system. We can think of the real world as being made up of objects and so we can think of a computer system as being made up of objects. For example a library has customers, books, DVDs, assistants etc and if we made up a software system for managing the library then the same objects would be involved in that system.

However, although the objects define the first level of separation in a software system, each object is assigned *behaviour* or a set of *services.* The services consist of a set of modules, hence modular programming is just as important (if not more important) in object-oriented programming.