Modules

A *module* is a collection of related items, which may include functions, types, constants and variables. A single function is the simplest example of a module. Modules that we have already been using include "*iosteam*", which has a collection of items to be used for input and output; also "*ctype.h*", which has a collection of items pertaining to character manipulation and testing.

In C and C++, a module is implemented via a pair of files whose names differ only in their extension. The two files are described below.

1) **Specification file**: A ".h" (header) file containing function prototypes, constant declarations, type declarations and various other items required by the module. It is generally considered unwise to include variable definitions in the specification file.

2) **Implementation file**: A ".cpp" file that contains source code for each of the functions prototyped in the specification file. The implementation file must not redefine the meaning of type definitions or constant values in the specification (header) file ("#ifndef" preprocessor directives can be used to prevent this.) An implementation file can be compiled to create a ".obj" (object) file that can be linked to a client's program or added to a library file, this is not a ".exe" file.

Separate compilations of modules have the following advantages:

1) One module can be used by many clients (or programs).
2) Individual modules can be modified and recompiled without recompiling the entire program.
3) Programs that use the module will compile faster, since the module code does not have to be recompiled.
4) The source code of the module can be kept hidden from the client, since all the client gets is the object code file to link with. The source code is therefore protected from those who should not change it.

Modules are important because they are **reusable** or considered **off-the-shelf** components. In programming classes we do most of our programming from scratch, but in the real world you use reusable code or modules as much as possible. This has several advantages:

1) Programmers do not have to reinvent the wheel.
2) It is less expensive than doing the work in-house.
3) Off-the-shelf modules are generally reliable.
4) Off-the-shelf modules generally perform better.

The design of modules should be done in such a way that the implementation of the module should be hidden from the user of this module. The design decisions should be kept as **local** as possible to module. The locality of private data, i.e. data only known within the module itself, and the algorithm within a module yield the following benefits:

1) This allows division of labor, separate teams to work on different parts of a project without fear of affecting other teams.
2) It allows a person to understand a module more easily, since knowledge of one module is possible without understanding a number of others.
3) It helps a module to remain independent, if changes are made, they do not affect other modules.

Remember a well designed function also exhibits this notion of locality. A classic blunder is to reference an external variable within a function.