Separate Compilation in C

There is no way for the compiler or the linker to detect inconsistencies among declarations or uses of a name in different files. The current rules can be summarized as follows:

• If the declaration of a global object (variable or function) contains the word `static`, the the object has **internal linkage**, and is identified with (linked to) any other internally linked declaration of the same name in the same file.

• If the declaration of a function does not contain the keyword `static`, then it has **external linkage**, and is identified with any other (non-static) declaration of the same function in any file of the program. (A function declaration may consists of just the header.)

• If the declaration of a variable contains the keyword `extern`, then the variable declaration has the same linkage as any visible, internally or externally linked declaration of the same name appearing earlier in the file. If there is no earlier declaration, then the variable has external linkage, and is identified with any other declaration of the same external variable in any file of the program. In other words, files in the same program that contain matching external variable declarations actually share the same variable. A global variable also has external linkage if its declarations syas neither `static` nor `extern`.

• If an object is declared with both internal and external linkage, the behavior of the program is undefined.

• An object (variable or function) that is externally linked must have a **definition** in exactly one file of a program. A variable is defined when it is given an initial value, or is declared at the global level without the `extern` keyword. A function is defined when its body (code) is given.

The "linkage" rules of C89 provide a way to associate names in one file with names in another file. The rules are most easily understood in terms of their implementation. Most language-independent linkers are designed to deal with **symbols**: character-string names for locations in a machine-language program. The linker's job is to assign every symbol a location in the final program, and to embed the address of the symbol in every machine-language instruction that makes a reference to it. To do this job, the linker needs to know which symbols can be used to resolve unbounded references in other files, and which are local to a given file. C89 rules suffice to provide this information. For the programmer, however, there is no formal notion of **interface**, and no mechanism to make a name visible, for example, to declare an external object variable as a multifield record in one file and as a floating-point number in another.The compiler is not required to catch such errors, and the resulting bugs can be very difficult to find.

Header files

C programmers have developed conventions on the use of external declarations that tent to minimize errors in practice. These conventions rely on the **file inclusion** facility of a macro preprocessor. The programmer creates files in pairs that correspond roughly to the interface and the implementation of a module. The name of an interface file ends with `.h`; the name of the corresponding implementation file ends with `.c`. Every object defined in the `.c` file is **declared** in the `.h` file. At the beginning of the `.c` file, the programmer inserts a directive that is treated as a special form of comment by the compiler, but that causes the preprocessor to include a verbatim copy of the corresponding `.h` file. This inclusion operation has the effect of placing "forward" declarations of all modules's objects at the beginning of its implementation file. Any inconsistencies with definitions later in the file will result in error messages from the compiler. The programmer also instructs the preprocessor at the top of each `.c` file to include a copy of the `.h` files for all of the modules on which the `.c` files depends. Unfortunately, it is easy to forget to recompile one or more `.c` files when a `.h` file is changed, this can lead to very subtle bugs. Tools like Unix's `make` utility help minimize such errors by keeping track of the dependences among modules.

Namespaces

Even with the convention of header fiels, C89 still suffers from the lack of scoping beyond the level of an individual file. In particular, all global names must be distinct, across all files of a program, and all libraries to which it links. Some coding standards encourage programmers to embed a module's name in the name of each of its external objects, but this practice can be awkward.

To address this limitation, C++ introduced a `namespace` mechanism that generalizes the scoping already provided for classes and fuctions, breaks the tie between module and compilation unit, and strengthens the interface conventions of `.h` files. Any collectio of names can be declared inside a `namespace`. Actual definitions of the objects can appear in any file. Definitions of objects declared in different namespaces can appear in the same file if desired.

A C++ programmer can access the objects in a namespace using **fully qualified** names, or by **importing** (using) them explicitly. There is no notion of export; all objects with external linkage in a namespace are visible elsewhere if imported or accessed with their qualified name. Note that linkage remains the foundation for separate compilation: `.h` files are merely a convention.