Modules 2024

I created a new Modules scheme a couple of years ago. Based on my experience since, this has been simplified.

This new version is in use in two languages, one lower-level systems language, one dynamic scripting language, both ahead-of-time compiled. Both are whole-program compilers. Although the comments mostly have the systems one in mind.

There is quite a lot to explain, but in brief: **all the modules comprising a program are listed at the top of the lead module**; that's pretty much it!

With this scheme, no separate build system is needed to turn a bunch of sources into an EXE or DLL file.

What is a Module

A 'module' in this scheme is always one source file. One module cannot be implemented across multiple source files. One source file cannot define multiple modules. Modules cannot contain other modules (those would be more like classes).

The name of a module must be both a valid identifier in the language, and a valid filename.

Example Pro

I will use an example program P consisting of 4 modules: P (the lead module), A, B and C. All information about the layout of project is given in the lead module. There are two ways do this:

(1) P contains only the list of other modules (it doesn't include itself) and no other code. So it looks like this:

```
module A
module B
module C
```

This is the pattern I use for most projects. This allows the lead module to be easily swapped with another, with a slightly different set of modules to provide an alternate configuration.

(2) P can also contain code, although here you'd probably dispense with P completely, and put the module info at the start of A:

```
module B
module C
.... the rest of module A ....
```

(Note that the application will now be called A, but of course you can name the modules P, B, C, or you can choose a name at compile-time.)

There is no project info, no `module` or `import` statements, in any other module. Other module schemes tend to have rag-bag collections of `import` statements at the top of every module, which in my view is unnecessary micro-managing.

The SubProgram

Modules in my scheme are grouped into SubPrograms. Within that group, any entity exported by any module (using a `global` attribute), is visible to all modules in the group. No specific `import` is needed, so long as all modules are listed in the lead module. My example program contains one subprogram.

No name-qualifier is needed either: to call a global function `F` defined in `B` from `A`, I can just write `F()`. I only need to write `B.F()` if, for example, `C` also exported a function `F`.

So all modules in the subprogram group are on familiar terms with each other. There is no hierarchy. There can be cycles: A can import B that can import A. (There is an ordering however; see below.)

Multiple SubPrograms

Most of my current projects have one subprogram - one group of chummy modules (plus the standard library; see below). Programs can have several subprograms, but each should be a group of modules that could be compiled by themselves, either into an EXE file, or into DLL file (when there is no `main` entry point and the subprogram is a library).

Suppose there is a 3-module library Q with modules Q, X, Y. Q might contain:

```
module X
module Y
```

To incorporate this into P, so that Q is statically compiled into the same EXE, P is defined like this:

```
module A
module B
module C
import Q           # read further modules from Q
```

The resulting program compromises modules P, A, B, C, Q, X, Y, although P and Q contain only module info here.

Visibility between SubPrograms

Even global entities between the modules of Q, for example, are not visible from P, unless they are specifically exported from Q. This involves using an `export` attribute instead of `global`. (It is not possible to export without also making a name global in that subprogram.)

So if X exports a function `G`, it can be called from module A using `G()`, you don't need to qualify the name unless there is again a clash. But if you do it will be written as `Q.G()` not `X.G()` or `Q.X.G()`; P knows nothing of the internal modules of Q. (It is not possible to export two different `G` functions from Q.)

Here there is a hierarchy of dependencies; cycles between subprograms are not allowed. The first subprogram (P in my example) is at the top of the hierarchy.

Q can't call exports of P, as it needs to work standalone.

The Standard Library

For my static language, this is a collection of 5 modules, listed in a 6th module called `msyslib`. This would normally require this line in each application:

```
import msyslib
```

But this module is included automatically, unless specifically excluded. The standard library is anyway special since the source files are expected to be embedded within the compiler, not be files on disk.

NameSpaces

Each module name creates a namespace within that subprogram. And each subprogram name also creates a namespace visible across the program. Mainly these are used for disambiguation when global or exported names clash. In that case, aliases can be created:

```
 module longmodulename as lmn
```

to keep accesses short. In the dynamic language, module names can be stored in variables and used for name resolving at runtime.

Creating a Library

Any program can be compiled to a DLL file (Windows dynamic shared library) rather than EXE. Any names with `export` attribute in the top or only subprogram, are also exported from the DLL library. (In this case, the names are unadorned. If `B.F` is exported, it will have the name `F`, so some care needs to be taken to avoid clashes.)

(Comments about EXE and DLL obviously refer to the systems language.)

In principle, anything in its own subprogram can be made into a DLL, and the same functions called via the usual FFI methods. If the DLL is created with my compiler, it will automatically produce an exports file to allow its use from my language. So if Q is compiled, it will create a module called Q_LIB. Then P can be revised to be this:

```
module A
module B
module C
module Q_LIB           # contains FFI module to access exported entities of Q
```

Library Imports

One other thing the scheme specifies is any external libraries that are needed to build the application. For example:

```
linkdll opengl
```

This is only needed (in my language) if there isn't an import module somewhere with an FFI block that explicitly names the library. (Often there is no clean link: an FFI block might named 100 functions which exist in three separate DLLs, or the exact DLL name depends on version which I want in one place. Sometimes multiple FFIs declare names from the same DLL.)

Names imported via FFI are given a global attribute. So I can have a module (say SDL) that imports a bunch of functions from SDL2 for example, those same functions are then visible to all other modules in the group. I just need `module SDL` in the lead module.

The File System

The module scheme tries to be independent of the file system. But it can't always manage that.

There is currently a weak spot: unless all input modules are in the same directory of the lead module, it needs to be told where to look. But as it's done now, that info is hardcoded within the project info, which is undesirable. (In general I will not know for sure where some arbitrary subprogram resides, or it cou;ld change.) See the real example below.

So this needs a better solution. Otherwise with that first example starting module P:

```
module A
module B
module C
```

This represents 4 source files, `P.m A.m B.m C.m` (assuming my systems language). The location of `P.m` depends on what path was provided to the compiler, so if invoked like this:

```
mm \abc\def\p                    # note both file system and language are case-insensitive, a
```

Then `P.m` is in directory `/abc/def/`, and the other modules are looked for there unless the path is overwridden as shown below.

In the case of the standard library modules, those source files are embedded inside the compiler. (There is an option to load them from disk, but it then looks somewhere that is only meaningful on my own machine as developer.)

In this scheme, the compiler will always look in exactly one place for a source file. It will never look in a range of places (that can lead to inadvertently mixing versions, or even loading an unrelated file of the same name).

Module Evaluation Order

A feature of my languages is that there is an optional special function `start` in each module. If present, this is automatically called when the program starts. (In the dynamic one, there can also be file-scope variables initialised with runtime expressions.)

This can be used to initialise various data and data structures. However, behaviour may rely on the order each function/each module is invoked. With a scheme using `import` everywhere, this can be unpredictable. Here, it is strictly in the order the modules are listed in the lead module, except the module containing the entry point (which must be near the start) is done last.

Program Entry Point

This is the function called `main` in the main subprogram, and specifically in the first module or the second. Other `main` functions in other modules are ignored.

(In the dynamic language, individual modules can be run directly. If there is a `main` function in the lead module submitted, it will call it. This is sometimes used for test code for that module.)

Real Example

This is from a C compiler, an old one which incorporates an x64 assembler as a separate subprogram. Thus there are two subprograms plus the standard library. This is lead module `cc.m`:

```
    module cc_cli
    module cc_decls
    module cc_blockmcl
    module cc_export
    module cc_genasm
    module cc_genmcl
    module cc_headers
    module cc_lex
    module cc_lib
    module cc_libmcl

    module cc_parse
    module cc_support
    module cc_tables

    $sourcepath "c:/ax/"
    import aalib
```

That subprogram has its own lead module which is this:

```
    module cc_assembler
    module aa_decls
    module aa_disasm
    module aa_genss
    module aa_lex
    module aa_lib
    module aa_mcxdecls
    module aa_objdecls
    module aa_parse
    module aa_tables
    module aa_writeexe
    module aa_writeobj
```

Normally, those `aa_` modules are used with a different lead module, `aa.m`, which is built into a standalone assembler:

```
    module aa_cli

    module aa_decls
    module aa_genss
    module aa_lex
    module aa_lib
    module aa_objdecls
    module aa_mcxdecls
    module aa_parse

    module aa_tables
    module aa_writeexe
    module aa_writemcx
    module aa_writeobj

#   module aa_disasm                # these two are optional; only used for development
#   module aa_writess
    module aa_writessdummy
```

The two projects are built like this:

```
c:\bcx>mm cc
Compiling cc.m to cc.exe

c:\ax>mm aa
Compiling aa.m to aa.exe
```

Directives

They are:

```
     module name [as name]
     import name
     $sourcepath string                  # (temporary feature until sorted)
     linkdll name
```