Modules

At present, modules are not supported in g++. So the rest of this document will be about using them in Visual Studio.

Whether we should care is addressed in the textbook, Chapter 27.

Modules in Visual Studio

First, let's look over a working program; after that let's go through the process of setting up your own project.

Coding with modules: an example

Example 1 shows a module written for Visual Studio. You can find it in source code's `ch27/modules` folder. It isn't in `ch27.sln` -- go into `ch27/modules` and open `modules.sln` instead.

I recommend you load it now and build it, because it's different from what we're used to. You can't just click Build or Run to build: for now at least you must right-click each `.ixx` (module) file and select Compile, in the order you want them compiled. *Then* you can build the project.

Now to look at the code. Example 1's line `export module greetings;` tells C++: this file should be exported as a module (named `greetings`).

Subsequent items are `export`ed if we want the rest of the world to see them.

Example 1. A module: `greetings.ixx`*. Visual Studio expects modules to end with* `.ixx`*.*

```
// Module for lists of hello, goodbye messages in different languages
//    -- for _C++20 for Lazy Programmers_

export module greetings;

namespace languages // Ordinarily I'd name this same as the module, but I want to
                    //  be clear when I'm referring to namespace and when I'm referring
                    //  to module, for this example
{
      export constexpr int NUM_LANGUAGES = 3;

      // Greetings in English, Espanol, Esperanto...
      export const char* helloMessage  [] = { "Hello",   "Hola",  "Saluton" };
      export const char* goodbyeMessage[] = { "Goodbye", "Adios", "Adiau"   };
```

```
    int secretFunction () { return 1; } // not visible to outside world
}; //namespace languages
```

Example 2 -- since it starts with `export module` -- is also a module file. But it also happens to import things. `import std.core;` imports the standard library (at least parts of it). Since this file needs things from the `greetings` module, it also says
`import greetings;`
...and since it wants those things also re-exported, we prepend `export` to that line:
`export import greetings;`

Example 2. `printGreetings.ixx`*: a module that imports another.*

```
// Module for printing greetings in various languages
//      -- for _C++20 for Lazy Programmers_

export module printGreetings;

import std.core;
export import greetings;

export void printHello   (int language)
{
    std::cout << languages::helloMessage  [language] << '\n';
}

export void printGoodbye (int language)
{
    std::cout << languages::goodbyeMessage[language] << '\n';
}
```

Finally, we have the file for `main`. It imports our module `printGreetings`; everything else should look familiar.

Example 3. A program that imports modules.

```
//Telling the world hello in different languages using modules
//  -- from _C++20 for Lazy Programmers_

import printGreetings;      // for printHello, NUM_LANGUAGES

int main()
{
    for (int lang = 0; lang < languages::NUM_LANGUAGES; ++lang)
        printHello (lang); // for each language, say hello

    return 0;
}
```

Output:

```
Hello
Hola
Saluton
```

That's it: we now know how to write simple modules and import them into our `.cpp` files.

Making your own project

Now, instructions on making your own.

After creating your project, go to Project Properties, then Configuration Properties > C/C++. Under Language, be sure you have C++ Language Standard set to /std:c++latest and Enable C++ Modules (experimental) set to Yes. See the source code, any project file, Project Properties, or `newWork/UsingC++20Features.txt`, for more.

Set Code Generation > Runtime Library to /MDd (for Debug) or /MD (for Release).

Setting General > SDL checks (nothing to do with SDL2) to <inherit from parent or project defaults> eliminates this warning:

```
C:\...\main.cpp(4,16): warning C5050: Possible incompatible environment while importing module 'std.core': _GUARDOVERFLOW_CRT_ALLOCATORS=1 is defined in current command line and not in module command line
```

I think warnings will go away as Visual Studio refines its handling of modules. Anyway, we can ignore them.

You'll need to add module files to your project. Visual Studio expects them to end in `.ixx`. After adding one, you have to tell Visual Studio this is the kind of file it should compile. Right-click it in Solution Explorer, and go to Properties > Configuration Properties > General. Set Excluded from Build to No and (important) Item type to C/C++ compiler (*not* C/C++ header).

Antibugging

- **Things you're using modules to define show up with squiggly red lines.** The editor's not always right; ignore those lines.
- **You try to compile a module (`.ixx`) file by right-clicking, but Compile is greyed out.** See "Making your own project" above: you have to tell Visual Studio this is a compilable file.
- **You keep hitting build, but it doesn't rebuild the `.ixx` files.** See the "Coding with modules" section above: you have to tell it to compile each such file.
- **`"Could not open output file '<something>.ifc'`.** Restart the compiler.

Summary

We're just not there yet. The plan is for to C++23 to support importing the standard library as modules; I hope, and expect, both compilers will be ready to fully support other aspects of modules by then as well. But now, thanks to Microsoft, you've had a foretaste.