C++ Modules

Peter Chapin

CIS-3012, C++ Programming

Vermont State University

Declaration vs Definition

- Technically, we distinguish between declarations and definitions
  - Many people don't worry about this distinction (or even know about it)
  - It's a useful concept, however.
- A *declaration* announces the existence of some entity.
- A *definition* provides an implementation of that entity.
- Definitions are also declarations!
  - That is, the set of definitions is a subset of the set of declarations

Some Examples

- Global objects:
  - `extern int x; // Declares that the int x exists.`
  - `int x = 0;    // Defines the integer.`
  - Global objects have default initial values. The `extern` reserved word is what signals that you are dealing with “only” a declaration.
- Global functions:
  - `extern int increment( int x );  // A declaration*.`
  - ```
    int increment( int x ) {
        return x + 1;
    }   // Defines the function by providing its body.
    ```

* For functions the **`extern`** reserved word is optional, and it is rarely used.
    ```

External References

- C++ (normally) requires that every entity be declared before it is used.
  - Exception: In the body of an inline class method defined in the class body, it is permitted to use names declared in the class's private section even if the private section is later.
- However, when compiling a *translation unit* (TU), the compiler does not need to know where declared entities are defined…
  - … or even if they are defined at all!
- The linker "resolves external references" by connecting usages of declared entities with their definitions.
  - *It's what linkers do*.

Classes

- Classes are a little special.
  - `class Example;   // Only declares the class.`
  - ```
    class Example {  // Defines the class.
        // methods…
        // private section…
        // etc.
    };
    ```
  - A class declaration (also called an *incomplete class declaration*) allows very limited use of the class:
    - Only pointers and references to the class can be declared; no class objects.
    - No methods can be used (compiler doesn't know what they are yet)
    ```

Classes, Continued

```
class Example {
public:
    int add_to_state( int value );
private:
    int state;
};

int Example::add_to_state( int value )
{
    state += value;
}
```

The *definition* of a class…

… contains *declarations* of its methods…

… which are *defined* elsewhere.

The One Definition Rule

- C++ has a *One Definition Rule* (ODR)
  - Roughly: *Every entity must have only one **definition** in the entire program*.
  - Every function and every global object can only be defined in a single TU (but they can be declared in many places).
- But there are problems with this simple statement of the ODR.

The Reality on the Ground

- Prior to C++ 2020…
  - Each TU is compiled in isolation of all others.
  - This means all entities that are used in a TU need to be declared (or defined) in that TU
- Solution(?): header files!
  - Put declarations in header files
  - #include header files into the TUs that need to see those *declarations*.
  - Leave the *definitions* in the implementation files (*.cpp).
  - Many TUs will see the same headers and the same declarations, but that does not violate the ODR. Each definition is in exactly one *.cpp file.

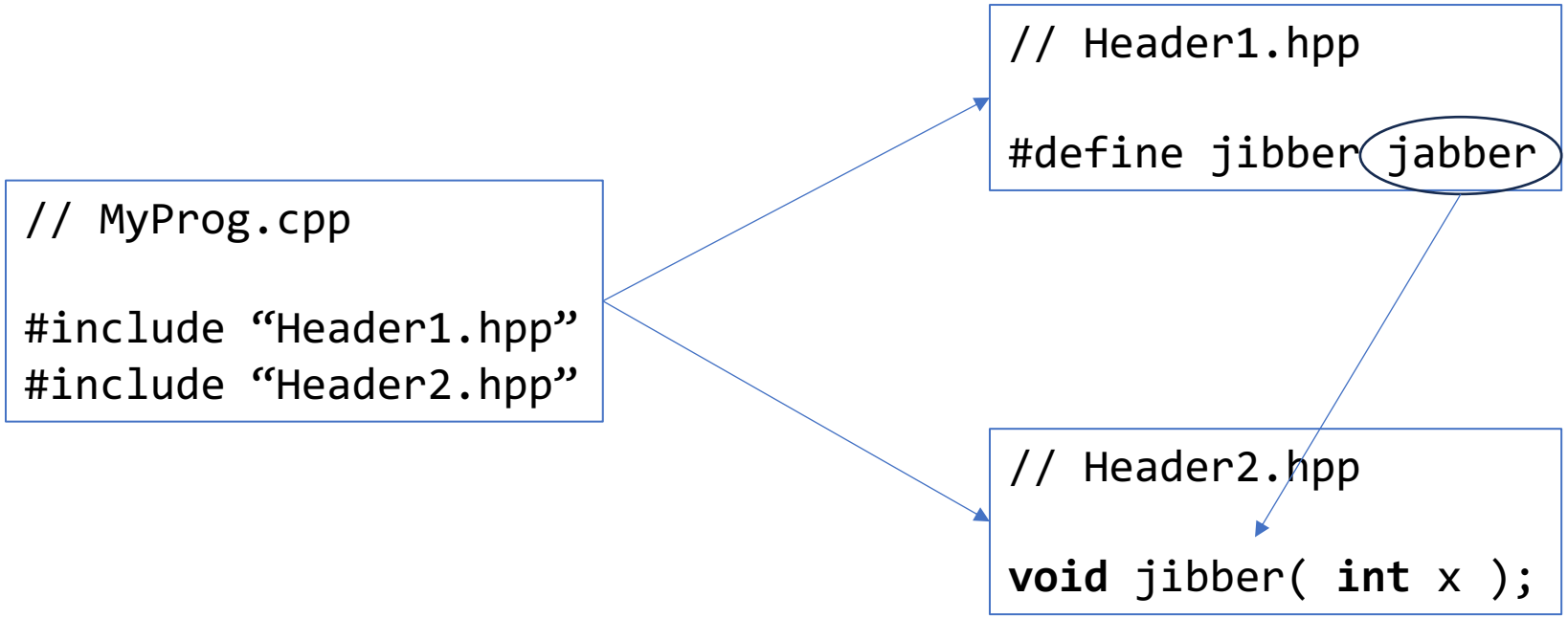
Unfortunately…

- Classes need to be defined in header files…
  - … so the compiler can know how much memory the objects consume
  - … so the compiler can know what methods they have
- Inline functions need to be defined in header files…
  - … so the compiler can expand their bodies at the call sites
- Templates need to be defined in header files…
  - … so the compiler can generate the needed specializations
  - C++ 98 had a feature called “exported templates” that was supposed to address this, but it was very difficult to implement, and almost no compiler vendor even tried. The feature was dropped in C++ 2011.

What About the ODR?

- Classes, inline functions, and templates all must be *defined* in header files, meaning that there is one definition in every TU that uses them. *Doesn't that violate the ODR?*
  - **Yes, it does**!
  - … but not really because: the description of the ODR in the standard is very long and complicated to allow for this reality.
  - Basically: It's okay if all definitions (of classes, inline functions, and templates) are "the same."
  - Sounds easy, right? If they are all in a common header file?
  - Not so fast…

Preprocessor Macros!



- The macro `jibber` bleeds into the other header and changes it…
- … even if the two headers are written by completely different people!

Contrived?

- The previous example is contrived, but the problem remains:
  - Macros in one header can edit code in other headers in arbitrary ways
  - *Macros don't respect scope* and don't know about namespaces!
- Various "best practices" have evolved to combat this problem
  - *Always* use `ALL_UPPER_CASE` names for macros
  - *Never* use `all_upper_case` names for anything else!
  - Don't use macros at all if you can possibly avoid them
- Since templates must be fully defined in header files, they are exposed to this problem much more than non-template code

The Other Problem with Headers

```
#include <map>
#include <set>
#include <string>
#include <vector>
```

All the code for all the methods is in the headers!
(yes, `std::string` is a template too)

- A huge amount of code is processed by the compiler *in every TU!*
  - The result is slower than necessary compilation times
- Consider: most of the standard library is made of templates
- Many third-party libraries are “header only” libraries, meaning they are all templates
  - Vendors must ship their libraries in source form (even if not open source)

Spurious Includes

- *Finish Me!*

Modules

- Starting with C++ 2020, there are now *modules* that deal with these problems
  - There are no header files, in the usual sense!
  - There is no #include!
- Wait… isn't that terribly incompatible?
  - Yes, but…
  - … it is permitted to keep using headers and #include as a transitional thing
  - … mixing traditional headers and modules is (theoretically) possible
  - … there are also things called *header units* which are midway between traditional headers and proper modules

Unfortunately…

- C++ 2020 does not require that the standard library be modularized
  - *It does not even require the vendor to provide header units* for the standard library* (although you can compile your own)
- C++ 2023 *does* require a standard module

```
// This is C++ 2023

import std;

int main( )
{
    std::vector<int> my_vector;
}
```

Makes the whole standard library visible

Still necessary to qualify names.
Module `std` and name space `std` are two different things!

* Header units will be discussed more in later slides

Modules Are Not Name Spaces

- In most languages, modules (or packages or whatever) are also used for name space control. This is not the case in C++
- C++ already has a name space feature (since 1998). Modules do not impose any restrictions or requirements on the name spaces
- `import my_module;`
  - This may (or may not) introduce a name space
  - The name space may (or may not) be the same name as the module
  - The module may (or may not) contain multiple name spaces
- If you care about name spaces (you do), you must use the existing mechanism inside the modules to arrange for them

So, What's the Point of Modules?

- Modules allow you to gather related code into separately compiled units that are isolated from each other
  - No bleed-over of macro names
  - No spurious includes in user translation units
- Modules are compiled once and for all
  - Importing a module is fast because the compiler does not need to parse and analyze the module code over and over for each TU that imports the module
  - This is why a single `import std` is reasonable despite the large size of the standard library!
- Modules offer some protection of source code (for closed source)

Sounds Great. How Does It Work?

- Instead of a traditional header file, you now write a *module interface unit*
  - … with a .ixx extension for Microsoft Visual C++
  - … with a .cppm extension for g++ and clang.
- The module interface unit contains essentially the same sort of information as a header file
  - Declarations of functions and global variables
  - Definitions of classes, inline functions, and templates
- The module interface unit can *also* contain definitions of functions!
  - Does not violate the ODR because it is only compiled once

Mixing Specification and Implementation?

- It is possible to put the entire contents of a module in the module interface unit, thereby mixing specification and implementation
  - Like what Java does in .java files
  - I am not a fan of this approach
- It is also possible (better) to put the definitions of the functions in a module into a separate .cpp file, distinct from the module interface.
  - Just as is done with the traditional .hpp/.cpp separation

Show Me Some Code

```
// This is a module interface unit.
// You can tell because it starts with “export module …”

export module number_theory;

// Declarations (or definitions) prefixed with “export” are usable outside the module
export bool is_prime( unsigned n );
export class RandomGenerator { … };
export inline bool is_even( unsigned n ) { return n % 2 == 0; }

// Declarations without “export” are for internal module use only.
unsigned gcd( unsigned a, unsigned b );
```

What Next?

- The module interface unit is compiled separately (and just once) to create the *built module interface* (BMI) file
- The format of the BMI file is compiler-specific and not portable or standardized. It is like the object files normally produced
- The BMI file contains the parsed and analyzed result of processing the module interface unit in a form that is easy for the compiler to ingest

Client Code?

- To use a module, import it:

```
import number_theory;
```

- All exported declarations are visible in the client program, much like including a header file. Except…
  - … preprocessing done in the module interface unit has no impact on the client code
  - … any imports in the module interface unit have no impact on the client code (the client must explicitly import all modules that are directly needed)
- Imports can be reordered with no semantic effect

What About Large Modules?

- A single module interface unit is a problem for very large modules
  - … like the module `std` defined by C++ 2023
- C++ 2020 modules can be *partitioned*
  - The top-level module interface file mentions the partitions
  - Separate files hold the declarations for the various partitions
- There is no semantic significance of the partitions
  - It's still one module
  - The partitions are only to make management of large modules easier
  - Don't forget that you can use name spaces to organize a large module's code base!

Module Implementation?

- In the common case where the definitions of the functions in a module are not in the module interface unit…
- … you can create one or more .cpp files with those definitions. Start with a module declaration:

```
module number_theory;
```

- Without the **`export`** keyword, this says the file is part of the named module's implementation

What about #include?

- There is no #include!
  - If you use modules throughout, you never need to #include again!
- Unfortunately, we must deal with mountains of legacy code
- The preprocessor is still active so you can still use it in module interface units and module implementation files in the usual way
  - It still does what it does, which is not very smart
  - Any macros #defined in a module interface unit are not made part of the BMI file, and thus can't be used by clients of the module. *This is one of the main points of modules!*

Header Units

- If you #include a header in a module interface unit, it is likely that none of the declarations in that header are preceded by **`export`**
  - Thus, those declarations are invisible to clients of the module, which is probably good (no spurious includes)
  - However, the compiler will assume all those declarations are internal to the module, which is probably bad
- C++ 2020 allows you to compile traditional header files into *header units*
  - The compiler creates a BMI file for the header as if all the (non-macro) declarations in the header were exported (even if not explicitly marked at such)

Importing Header Units

- To use a previously compiled header unit, do:

```
import <legacy_header>;
```

- The compiler looks for the BMI file *you* compiled from the original header
- This is useful for libraries that are not yet modularized (i.e., pretty much all existing libraries)
- This creates fewer problems than trying to #include the headers themselves

C++ 2020

- Unfortunately, C++ 2020 does not provide a modularized library and does not come with pre-compiled header units for the standard library
  - You must compile the header units yourself for all standard library headers you want to use. This is a major pain
- Solution: *Move to C++ 2023 at your first opportunity* so you can import module `std` directly. Use header units only for third party library or legacy code
- The “big three” compilers all agree to support importing module `std` in C++ 2020 mode also, but that’s not widely available yet