CHAPTER 11

Modules and Namespaces

While cramming all your code into a single source file is always possible (even for huge programs), a larger code base is far easier to manage if you organize related source code (functions, constants, types, and so on) into logical, composable units, each in their own files. Not least because you and others can then construct different applications out of these same building blocks. As of C++20, the composition unit of choice is a module.

The more your code base grows and/or the more third-party libraries you rely on, the more likely it becomes to have two functions, two global variables, two types, and so on, with the same name. Perhaps two independent subsystems inadvertently define their own log() or validate() functions, for instance, or their own ErrorCode enumeration types. Because real-life code bases easily span millions of lines of code, with each subsystem mostly developed by semi-autonomous teams, it is virtually impossible to avoid name clashes or name collisions without some agreed upon hierarchical naming scheme. In C++, the recommended approach is that different subsystems declare their names in disjoint namespaces.

In this chapter, you’ll learn

- How to build composable units of code in the form of modules
- How to export entities from a module so that they can be used in any source file that imports that module
- How to split larger modules into smaller, more manageable units
- The difference between module partitions and submodules
- More about using namespaces and how to define and manage your own
- The relation between modules and namespaces
- How to split a component’s interface from its implementation at the level of both modules and namespaces

Modules

At the lowest level of granularity, any C++ program is pieced together from elementary, reusable entities such as functions, types, and variables. And for really small programs—such as the ones you wrote so far—that may be all you need within your own code. But as your code base starts growing, you’ll soon notice that you also need some other mechanism to further organize your code at a higher level of granularity. You’ll want to group related functionality together, so that you and everyone else knows where to look for it. You’ll want to decompose your program into larger, discrete subcomponents, where each component solves one specific piece of the puzzle. Internally, these subcomponents can then use whatever combination
of functions, types, and/or other components they may need. As a user of a subcomponent, you do not even need to know how it works internally. In C++20, the mechanism of choice is to form self-contained subcomponents of related functionality, called modules.

A module can export any number of C++ entities (functions, constants, types, and so on), which you can then use in any source file that imports that module. A module may even export entire other modules. The combination of all entities that a module exports is called the module interface. Entities that are not part of the module’s interface (directly or indirectly) can only be seen and used from within the module itself.

You of course already know how this works at the consuming end of this relation. For instance, once you import the `<string>` module of the Standard Library, you know that you gain access to types such as `std::string` and `wstring`, and functions such as `std::getline()`, `stoi()`, and `to_string()`. The `<string>` module is said to export these entities; you, the consumer of the module, imports them.

In this chapter, you will learn how to create your own modules. You typically use a separate module for each collection of code that encompasses a common purpose. Each module would represent some logical grouping of types and functions, together with any related global variables. A module could also be used to contain a unit of release, such as a library. Modules thus come in all shapes and sizes. Some essentially export only a single type, a single template (`<vector>`, `<optional>`, `<span>`, and so on), or some constants (the `<numbers>` module), whereas larger modules may export a vast library of related types and functions (the standard `<filesystem>` module is a good example of a somewhat larger module). You are free to choose the granularity of your modules.

As some of your modules inevitably grow larger and/or more complex, you may want to further decompose them again into smaller, more manageable subcomponents. We will show you how you can then either decompose such modules internally (using module partitions), and/or split them into smaller, loosely related modules (which are then often referred to as submodules). But before we get to organizing larger modules, of course we’d better make sure that you can create a smaller one first.

---

**Caution** At the time of writing, there was no compiler yet that fully supported modules. Some compilers already had experimental support of this C++20 language feature, but this was still very limited and unreliable. While we took great care in researching the language’s specification, we could therefore not adequately try out most module-related code in this and subsequent chapters. If your compiler does support modules and something does not seem to work, you should therefore consult the errata on the book’s website first. Also, if your compiler still does not support modules, know that the book’s online download section will contain equivalents of all examples and exercise solutions that do not use modules. If you are still forced to use nonmodular code, you may then also want to read Appendix A first after this chapter, and before continuing with the remainder of the book. Appendix A is available online.

---

1 Because `<string>` predates C++20, it is not technically written as a module but as a header. It therefore also does not actually export any entities the way a true module would—at least not explicitly. Most C++ Standard Library headers, though—all except those that it inherits from C, such as `<cmath>` and `<cctype>` (see also Chapter 2)—can be treated and imported as if they were a module. All entities that these headers define then act as if they were exported from a module. Conceptually, it is therefore perfectly fine to think of `<string>` and most other Standard Library headers as a module. We refer to Appendix A for further details on the relation between headers and modules.
Your First Module

This next file contains the definition of your first module. It is a module named math that exports some features that are glaringly missing from the standard <cmath> header: A template for functions that square a number (deemed too advanced for <cmath>, which therefore offers only the inverse, std::sqrt()), the lambda constant (our homage to one of COVID-19’s smartest victims: math’s most playful genius, John Horton Conway), and a function to determine the oddity of an integer (a notoriously hard math problem).

```cpp
// math.cppm - Your first module
export module math;

export auto square(const auto& x) { return x * x; }  // An abbreviated function template

export const double lambda{ 1.303577269034296391257 };   // Conway’s constant

export enum class Oddity { Even, Odd };  
bool isOdd(int x) { return x % 2 != 0; }  // Module-local function (not exported)
export auto getOddity(int x) { return isOdd(x) ? Oddity::Odd : Oddity::Even; }
```

**Note**  At the time of writing, there is no consensus yet on what file extension to use for module files. Each compiler, for as far as they already support modules, uses its own conventions. In this book we use the .cppm extensions for module interface files (.cppm is short for C++—cee plus plus—module). Consult your compiler’s manual to determine how to best name your module files.

Besides the use of some new keywords\(^2\), export and module, and of course the lack of a main() function, this module file looks exactly like any other source file you have seen thus far. In fact, the syntax for creating a basic module is so straightforward that it almost needs no further explanation. Except for perhaps the module declaration on the first line, that is...

Somewhere at the start of every module file\(^3\) is a module declaration. Every module declaration contains the module keyword followed by the module's name (math in our example). If the module keyword is preceded by the export keyword, the module file is a module interface file. Only module interface files may contribute exported entities to the module's interface.

The rules for naming a module are mostly the same as for naming any other entity in C++, except that within a module name you may use dots to string together multiple identifiers. Module names are the only names in C++ in which this is allowed. So, not only are math123 and A_B_C valid module names, but so are math.polynomials and a.b.c. Examples of invalid names include 123math, .polynomials, abc., and a..b. We explain later how you can use dotted names to simulate, in essence, hierarchical submodules.

\(^2\) Modules were only introduced in C++20. While export has always been a C++ keyword, module and import were not. To avoid breaking existing code, these two could not simply be turned into full-blown keywords like export, switch, or enum. That is, you are, technically, still allowed to use module or import as the name of a variable, function, type, or namespace. Of course, we strongly advise against doing so in new code. If you ever do have to use legacy entities named either module or import in module files, you may have to add the scope resolution operator :: (introduced later in this chapter) in front of their names at times to make the compiler accept your code (that is, you then may have to use ::module or ::import instead of module or import).

\(^3\) When reading about modules, you may also encounter the term module unit instead of module file. We refer to Appendix A for the difference between translation units and source files.
To export an entity from a module, you simply add `export` in front of its first declaration. In `math.cppm` you see an example of an exported template, an exported variable, an exported enumeration type, and an exported function, `getOddity()`. A second function, `isOdd()`, is purposely not exported. Only module interface files may contain `export declarations`, and only after their `module declaration`. Only entities that are exported by a module can be used in files that import the module.

Here is how you import the `math` module defined by `math.cppm` and use its exported entities.

```cpp
// Ex11_01.cpp – Consuming your own module
import <iostream>;
import <format>;
import math;

int main()
{
    std::cout << "Lambda squared: " << square(lambda) << std::endl;

    int number;
    std::cout << "Please enter an odd number: ";
    std::cin >> number;
    std::cout << std::endl;

    // if (isOdd(number))            /* Error: identifier not found: 'isOdd' */
    //   std::cout << "Well done!" << std::endl;

    switch (getOddity(number))
    {
        using enum Oddity;
        case Odd:
            std::cout << "Well done! And remember: you have to be odd to be number one!";
            break;
        case Even:
            std::cout << std::format("Odd, {} seems to be even?", number);
            break;
    }
    std::cout << std::endl;
}
```

**Note** Before you can compile a file such as `Ex11_01.cpp`, you generally have to compile the interface files of all modules that it imports, which in this case includes that of the `math` module. Compiling a module interface creates a binary representation of all exported entities for the compiler to quickly consult when processing files that import the module. How to compile modules again varies between compilers. Consult the documentation of your compiler for further details.

You import the `math` module by the obvious `import math;` declaration. Unlike with `import declarations` for Standard Library modules, you do not put angle brackets around the name of your own modules. (We refer you to online Appendix A for more information about the use of angle brackets in `import declarations`.)
Note Unlike for header files (see Appendix A), the name that you use to import a module is not based on the name of the file that contains the module interface. Instead it is based on the name that is declared in the export module declaration. While some compilers require the filename to match the name of the module, other compilers allow module interface files with arbitrary names as well. So, even though in this book we always name our module interface files module_name.cppm, do bear in mind that the name of this file is not technically what determines the name you should use in your import declarations.

Once a module is imported, you can use all entities that it exports. The `main()` function of Ex11_01, for instance, uses all entities exported by our grand and powerful math module: an instantiation of the `square<>()` function template, the `lambda` constant, the `getOddity()` function, and the `Oddity` enumeration type.

The key, though, is that you can only use these exported entities. As an example, you could try to uncomment the if statement that calls `isOdd()` in Ex11_01.cpp. Your compiler will then signal that it does not know of any function named `isOdd()`. And there is nothing odd about that: It is not because `isOdd()` is declared in the module’s interface file that it automatically becomes part of the module’s interface. To make `isOdd()` visible from outside the `math` module, you have to add `export` in front of its definition as well.

There is very little new otherwise to the `main()` program of Ex11_01.cpp, except for maybe the using enum declaration in the switch statement. We showed you how to use using enum declarations in Chapter 3, but this is the first time we used one in a switch statement (introduced in Chapter 4). Thanks to this using declaration, you can omit `Oddity::` from before the enumerator names `Even` and `Odd`—yet only within the scope of that switch statement. Later in this chapter, you will see that you can similarly put other using declarations and using directives at the scope of statements, functions, or namespaces as well.

Export Blocks

In Ex11_01, we exported four different entities, one by one, using as many individual export declarations. You can also export multiple entities all at once by grouping them inside an export block. An export block consists of the export keyword followed by a sequence of declarations between curly braces. Here is an alternative definition of the `math` module of Ex11_01 that uses an export block instead of a sequence of four export declarations (this module file is available from Ex11_01A).

```cpp
// math.cppm – Exporting multiple entities at once
export module math;

bool isOdd(int x) { return x % 2 != 0; }     // Module-local function (not exported)

export
{
  auto square(const auto& x) { return x * x; }

  const double lambda{ 1.303577269034296391257 };    // Conway’s constant

  enum class Oddity { Even, Odd };    
  auto getOddity(int x) { return isOdd(x) ? Oddity::Odd : Oddity::Even; }
}
```
Separating Interface from Implementation

So far, all entities were always defined directly in the module interface file. And for small functions and types such as those of our mighty `math` module, that works just fine. Sure, `math`'s interface file is perhaps a bit cluttered already by the module-local `isOdd()` function, but that is not too distracting yet. In real life, however, function definitions may span dozens of lines each, and you may need several more local functions and types to realize your exported interface. If you’re not careful, all these implementation details can obfuscate the module’s interface, making it harder for someone to get a quick overview of what a particular module has to offer. In such cases, you should consider separating the module's interface (containing for instance only the prototypes of exported functions) from its implementation (which is then where these exported functions are defined, together with any module-local entities).

Your first option to separate a module’s interface from its implementation is to do so within the module interface file itself. This is shown here:

```cpp
// math.cppm - Exporting multiple entities at once
export module math;

export // The module's interface
{
    auto square(const auto& x);

    const double lambda = 1.303577269034296391257; // Conway's constant

    enum class Oddity { Even, Odd };  // Cannot be moved.
    auto getOddity(int x);
}

// The implementation of the module's functions (+ local helpers)
auto square(const auto& x) { return x * x; }

bool isOdd(int x) { return x % 2 != 0; }
auto getOddity(int x) { return isOdd(x) ? Oddity::Odd : Oddity::Even; }
```

You can always move function definitions to the bottom of the module interface file like this. You could even move the definition of the enumeration type `Oddity` down, after first declaring it using `enum class Oddity;`. We left `Oddity`'s definition in the `export` block, though, because it is—and needs to be—part of the module’s interface (consumers need to know the enumerators to use the module). Constants have to be initialized immediately, so the definition of `lambda` cannot be moved.

Note that you knew most of this already, though, from earlier examples, where we often put all function prototypes at the top of a source file as well (to get a nice overview of the available functions), and all function definitions at the bottom (often after the `main()` function in which they were used).

The only truly new thing worth noting at this point is that you do not have to repeat the `export` keyword in front of the definitions at the bottom. You can, but you do not have to. That is, you can repeat `export` in front of the definitions of `square()` and `getOddity()` at the bottom of `math.cppm`, but doing so is entirely optional.

What is not allowed, however, is to add `export` in front of the declaration of an entity that was first declared without `export`. The following sequence of declarations, for instance, would therefore result in a compilation error.

```cpp
bool isOdd(int x);
// ...
export bool isOdd(int x); /* Error: isOdd() was previously declared as not exported! */
```
Module Implementation Files

Instead of putting all your definitions at the bottom of the module interface file, you can typically move them into a separate source file as well (we discuss some limitations in an upcoming subsection). The module interface file then includes the prototypes of all exported functions, while their definitions along with any module-local entities are moved to one or more module implementation files. This approach accentuates a clean separation of interface and implementation—a desirable trait of any software component.

Taking a small break from math modules, here is a slimmed-down module interface file for a module that offers functions to convert between unsigned integers and standard Roman numerals.\footnote{Don’t worry if you are not that familiar with Roman numerals: This section is about understanding the roman module, and not so much about understanding the implementations of the to_roman() and from_roman() functions.}

```cpp
// roman.cppm – Interface file for a Roman numerals module
eexport module roman;
import <string>;
import <string_view>;

export std::string to_roman(unsigned int i);
eexport unsigned int from_roman(std::string_view roman);
```

While naturally we have no room in this book to lay out truly large modules, we’re sure you can appreciate how this single interface file now gives a succinct, clear overview of what the roman module has to offer, free from any clutter caused by local helper functions or other implementation details.

New also in this module file is that it uses other modules—the <string> and <string_view> modules of the Standard Library to be precise. The main thing that you need to know about this for now is that in a module file, all import declarations must appear after the module declaration and before any other declaration. The following module file layout is therefore not allowed for the two annotated reasons:

```cpp
import <string>; /* Error: no imports allowed before the module declaration! */
eexport module roman;

export std::string to_roman(unsigned int i);
import <string_view>; /* Error: illegal import after declaration of to_roman()! */
eexport unsigned int from_roman(std::string_view roman);
```

With the roman.cppm module interface file in place, we can now start defining the module’s functions. Usually, for a small module such as roman, you would never go further than putting both function definitions into a single module implementation file—say roman.cpp. But just to illustrate the possibility, we will take things one step further and give each function definition its own implementation file. Here is the implementation file that provides the definition of the to_roman() function:

```cpp
// to_roman.cpp – Implementation of the to_roman() function
module roman;

std::string to_roman(unsigned int i)
{
    if (i > 3999) return {}; // 3999, or MMMCMXCIX, is the largest standard Roman numeral
    static const std::string ms[] { "", "M", "MM", "MMM" };
```
static const std::string cds[] = { "", "C", "CC", "CCC", "CD", "DC", "DCC", "DCCC", "CM" };  
static const std::string xls[] = { "", "X", "XX", "XXX", "XL", "LX", "LXX", "LXXX", "XC" };  
static const std::string ivs[] = { "", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX" };  
return ms[i / 1000] + cds[(i % 1000) / 100] + xls[(i % 100) / 10] + ivs[i % 10];  
}

Like all module files, a module implementation file contains a module declaration. Unlike module interface files, however, the module declaration of a module implementation file does not begin with the export keyword. Naturally, the implication is that a module implementation file may not export any entities. In fact, you are not even allowed to repeat the export keyword here in front of the definition of to_roman(); you can only add or repeat export in module interface files.

The implementation of the to_roman() function itself contains some clever, compact coding to convert unsigned integers into standard Roman numerals. Having mastered C++ expressions and functions in the first half of this book, we’re sure you can work out how it works, though. As the focus of this section is on modules, we’ll thus leave deciphering this function body as an optional exercise for you.

For completeness, here is the second module implementation file for our roman module, which as you know was so massive it could not possibly fit into a single implementation file.

// from_roman.cpp – Implementation of the from_roman() function
module roman;

unsigned int from_roman(char c)
{
    switch (c)
    {
        case 'I': return 1;  case 'V': return 5;  case 'X': return 10;
        case 'L': return 50; case 'C': return 100; case 'D': return 500;
        case 'M': return 1000; default: return 0;
    }
}

unsigned int from_roman(std::string_view roman)
{
    unsigned int result = 0;
    for (size_t i = 0, n = roman.length(); i < n; ++i)
    {
        const auto j = from_roman(roman[i]);  // Integer value of the i'th roman digit
        // Look at the next digit (if there is one) to know whether to add or subtract j
        if (i + 1 == n || j >= from_roman(roman[i + 1])) result += j; else result -= j;
    }
    return result;
}

This second implementation file is similar to the previous one, except that it defines a local helper function to convert a single Roman digit into an integer. This overload of from_roman() is only visible within this source file.

Because switch, for, and if statements are all familiar territory, we again have every faith that you can work out how these from_roman() functions work (should you be interested). We’ve added some comments to get you started (it’s never a bad idea to add such comments in your own code as well, whenever the logic is less than obvious).
Here is a small test program and its output:

```cpp
// Ex11_02.cpp
import <iostream>
import <string>
import roman

int main()
{
    std::cout << "1234 in Roman numerals is " << to_roman(1234) << std::endl;
    std::cout << "MMXX in Arabic numerals is " << from_roman("MMXX") << std::endl;
}
```

1234 in Roman numerals is MCCXXXIV
MMXX in Arabic numerals is 2020

**Note** With nonmodular code (see online Appendix A for more details), there was another, arguably stronger, motivation to not define functions in a header file (which is somewhat akin to a module interface file): every single change to a header, no matter how small (even fixing a typo in a code comment), generally meant that all code consuming the header (directly or indirectly) had to be recompiled. Worst case, due to recursive `#includes`, that meant rebuilding nearly the entire application over and over during development. Given that the implementation of a function typically changes far more frequently than its prototype, it therefore paid off to consistently move all implementations out of the header file and into a separate source file. Modules, however, do not suffer from this problem. Modules by design are self-contained, independent building blocks. Only changes to the actual module interface impact consumers, and function definitions are purposely *not* part of the module's interface\(^5\) (as discussed in the next subsection). You can thus safely change a function definition in a module interface file, even that of an exported function, without having to worry about recompiling any of the consuming code.

**Limitations to Implementation Files**

In general, everything that an importer of a module may need to consume an exported entity must be present in a module interface file. The reason is that whenever your compiler processes a source file that consumes that module, it only considers (the precompiled binary representation of) that module's interface. The output of compiled module implementation files is not considered at that time, and instead is only required for the linking stage (linking was briefly explained in Chapter 1). Some entities therefore must always be defined in a module interface file, and not in a module implementation file.

---

\(^5\) Unless that function is declared as `inline` with the `inline` keyword. Definitions of inline functions, like those of templates, *are* part of the module interface. We refer to Appendix A for a more detailed discussion of inlining and inline functions.
The definitions of all exported templates, for instance, must be part of the module interface. The compiler needs these template definitions to instantiate new template instances for the module’s various consumers. You effectively export the entire template, the complete blueprint for future instantiations, and not one particular function prototype (or type definition; see Chapter 17).

All the compiler needs to invoke a regular function, on the other hand, is its prototype. From that it knows what arguments to expect from consumers, and what the resulting type will be. Most function definitions can therefore be moved to the module interface file as is. Most, but not all. One example6 of a function definition that you cannot simply move out of an interface file is that of an exported function with auto return type deduction. For auto return type deduction to work, the compiler needs the function definition to be part of the module interface.

In the `math.cppm` interface file of Ex11_01, the `isOdd()` function is therefore the only entity whose definition you could move to a module implementation file as is. Not because `isOdd()` is the only entity that is not exported, but because `square<>()` is a template (thanks to its `const auto&` parameter; see Chapter 10), and because `getOddity()` uses auto return type deduction.

**Implicit Imports in Implementation Files**

Being the astute observer that you are, you of course noticed that we didn’t import the `<string>` module in `to_roman.cpp`, even though the definition of `to_roman()` clearly uses the `std::string` type. And, similarly, we didn’t import `<string_view>` in `from_roman.cpp` either. We invite you to take a quick look back if you missed that. The reason that we can omit these module imports is twofold.

- Every module implementation file (that is not a partition file; see later) implicitly imports the module that it belongs to. You can think of this as if a declaration such as `module roman;` implicitly adds `import roman;` on the next line. (Explicitly adding `import roman;` after `module roman;` is not allowed, by the way.)

- Whenever you import one part of a module into another part of the same module (more on this later), the latter essentially gains access to all declarations of the former7, even to declarations that are not exported. This includes not only declarations of module-local functions, types, and variables, but also imports of other modules, such as `import <string>;` and `import <string_view>;`.

Combined, these two rules entail that our module implementation files `to_roman.cpp` and `from_roman.cpp` both inherit the `import` declarations for `<string>` and `<string_view>` from the implicitly imported `roman.cppm` module interface file.

---

**Caution** Only module declarations of the exact form `module my_module;` add an implicit `import my_module;` declaration (and consequently prohibit explicit `import my_module;` declarations). Module declarations for module partitions, as seen later, carry no such implications. To gain access to all declarations of the module interface file in partition files, you have to add `import my_module;` explicitly.

---

6 Other examples include inline functions (whose definitions are required to be inlined at the calling sides) and constexpr functions (whose definitions are required for static evaluation). We discuss inline functions in more detail in Appendix A (available online).

7 This is actually not entirely accurate; entities with internal linkage can never be used from outside the file that they are declared in, not even within the same module. But since creating entities with internal linkage is never really required when writing modules (in technical speak, it is far easier to simply give all nonexported entities module linkage), we do not discuss that possibility here yet. Refer to Appendix A for more details about the different types of linkage.
Reachability vs. Visibility

When you import a module into a file that is not part of the same module, you do not implicitly inherit all imports from the module interface file. This is essentially why we had to import the <string> module in Ex11_02.cpp, even though the module interface file of the imported roman module contained an import declaration for <string> as well. You can, if you want, explicitly export import declarations from a module—we discuss this option further in an upcoming section—but by default, when you import a module, you do not implicitly gain complete access to all other modules that that module relies on.

And that is a good thing. That is, you really do not want all imports in interface files to be exported to consumers willy-nilly. You never want consumers of a module to be affected when that module changes the implementation of its interface. Only changes to a module's interface are allowed to impact its consumers. If all imports were exported and imported willy-nilly, any import declaration that you add or remove would ripple through recursively to all code importing that module, either directly or indirectly. And that could most definitely affect consumers; removing an import somewhere deep down could then break other code that came to rely on this import, while adding an import declaration could introduce name clashes. (Headers suffer from these exact problems; see online Appendix A for details.)

However, if you give consumers no knowledge about the types that are used in a module's interface, they mostly cannot use that interface anymore. Take the interface of the roman module of Ex11_02, for instance. To invoke from_roman(), you need to know how to create a std::string_view object; and, similarly, to use the result of to_roman(), you need to know at least something about the std::string type.

The fitting compromise that C++ modules offer is to distinguish reachability of an entity's declaration from the visibility of its name. We'll clarify what this means with an example. Copy all files of the roman module of Ex11_02 and start a new source file with a blank main() function. In this new source file, you should no longer import the <string> module like we did earlier in Ex11_02.cpp. The new source file, in other words, only contains these two import declarations:

```cpp
// Ex11_03.cpp – Using types with reachable definitions but whose names are not visible
import <iostream>
import roman;

int main()
{
   // You can put all code used in the remainder of this section here...
}
```

Any declaration (including type definitions) that is reachable in an imported module interface file such as roman.cppm (either through local declarations or through imports) is reachable in the importing file as well. By importing roman, the definitions of both std::string and std::string_view therefore become reachable in Ex11_03.cpp, for instance. Reachability of declarations thus spreads implicitly and recursively to any and all direct and indirect consumers of a module.

Once the definition of a type is reachable, you can, for the most part, use values of that type exactly the way you are used to. Because the definition of std::string_view is reachable in Ex11_03.cpp, for instance, you can invoke from_roman() there with a string literal, thus implicitly creating a std::string_view object in the process:

```cpp
std::cout << "MMXX in Arabic numerals is " << from_roman("MMXX") << std::endl;
```
You can even invoke functions on objects whose type definition is reachable. In formal speak, we say that the names of the members of a reachable class definition are visible. In the following two lines for Ex11_03.cpp, we can therefore invoke the c_data() and size() functions on the std::string object returned by to_roman(), even though we never imported the <string> module:

```cpp
std::cout << "1234 in Roman numerals is " << to_roman(1234).c_data() << std::endl;
std::cout << "This consists of " << to_roman(1234).size() << " numerals" << std::endl;
```

To avoid inadvertent name clashes, and to avoid consumers relying on varying internal names, the visibility of an entity's name does not escape a module willy-nilly. The std::string and std::string_view names are thus not visible in Ex11_03.cpp, and can therefore not be used there.

```cpp
// std::string_view s{ "MMXX" };   /* Error: the name std::string_view is not visible */
// std::string roman{ to_roman(567) };  /* Error: the name std::string is not visible */
```

Luckily, though, you do not need access to the names of the return types of functions in order to use them. You can always capture a function's result using auto or const auto& as placeholders for that name.

```cpp
auto roman{ to_roman(567) };
std::cout << "567 in Roman numerals is " << roman.c_data() << std::endl;
```

Like the visibility of type names (std::string, std::string_view, and so on), the visibility of function names does not escape a module's borders willy-nilly either. Functions such as std::stoi() and std::to_string() can therefore simply not be invoked from within Ex11_03.cpp.

```cpp
// std::cout << "std::stoi() is not visible: " << std::stoi("1234") << std::endl;
```

This also explains, at least in part, why we had to import the <string> module earlier in Ex11_02.cpp. Without that import, the following line does not work because the << operator is not visible. As you’ll learn in Chapter 13, this stream output operator is actually nothing more than a function as well, only one that can be invoked using a different syntax.

```cpp
// std::cout << "1234 in Roman numerals is " << to_roman(1234) << std::endl;
```

In summary, reachability of declarations implicitly and transitively propagates to all consumers of a module—however indirect—but visibility of declared names does not. Luckily, though, reachable definitions typically suffice for you to use a module’s interface without having to import any extra modules, especially if you embrace the use of auto.

**Exporting Import Declarations**

Like with most declarations in a module interface file, you can also add export in front of an import declaration. Any file that imports a module implicitly inherits all import declarations that are exported from that module as well. For instance, suppose in roman.cppm of Ex11_02 you add export in front of its two import declarations as follows:

```cpp
// roman.cppm
export module roman;
export import <string>;
export import <string_view>;
```
export std::string to_roman(unsigned int i);
export unsigned int from_roman(std::string_view roman);

Then you can safely remove the import declaration for <string> in Ex11_02.cpp, because that file then imports <string> indirectly by importing roman.

You should not systematically export all module imports that you use within a module, though. You should only export an import declaration if you know for a fact that every consumer of a module will need the functions and/or the names of the types of that dependent module to be visible. As you’ll recall from the previous section, reachability of type definitions is mostly sufficient to use a module’s interface, and for that you do not need to export the corresponding imports.

Another viable reason to export import declarations is to form so-called submodules. We discuss that strategy in the next section.

Managing Larger Modules

The larger a module becomes, or the more complex its functionality, the more the need may arise to further divide its code into smaller, more manageable components.

Of course, your first option then is to split your module into several smaller modules. This works well if its interface can be split into clean, disjoint subinterfaces, whose implementations then are independent as well. One particularly interesting approach then is to split larger modules into several so-called submodules.

We discuss that option in the first subsection.

Sometimes, however, you do want to keep everything bundled in a single, self-contained module. Maybe it makes no sense to split up its functionality. Or maybe the module interface is already quite small, and you just need a lot of internal machinery to realize it. Not all code needs to be exported to the entire application all the time. Keeping code local within a module has the advantage that you have more flexibility in changing it, without having to worry about external uses. The answer C++20 provides for such occasions is module partitions. We discuss partitions in a second subsection shortly.

Simulating Submodules

Even though C++ does not formally support the concept of nested submodules, you can simulate them quite easily. The principle is probably best explained by example. By creating the following three module interface files (we’ll omit the function definitions for brevity), you could split the colossal roman module into two smaller submodules, called roman.from and roman.to.

// roman.cppm – Module interface file of the roman module
export module roman;
export import roman.from; // Not: 'export import .from;' (cf. partitions later)
export import roman.to;

// roman.from.cppm – Module interface file of the roman.from module
export module roman.from;
import <string_view>;
export unsigned int from_roman(std::string_view roman);

// roman.to.cppm – Module interface file of the roman.to module
export module roman.to;
import <string>;
export std::string to_roman(unsigned int i);
It is important to note that, as far as C++ is concerned, `from` and `to` are just module names like any other. If you’d like to rename these (sub)modules to `from_roman` and `to_roman`, or even `cattywampus` and `codswallop`, by all means. The language does not enforce any hierarchical naming scheme at all. It’s just that adopting one makes it easier to see the relation between modules and its submodules, and dots were specifically allowed in module names to facilitate such hierarchical naming.

With these three module files in place, you have the following choice in the remainder of the application: either you import individual submodules, or you import them all at once. If you only need to output roman numerals, you can add `import roman.to;` and gain access to `to_roman()` only. Or you simply add `import roman;` and gain access to all submodules at once.

### Tip
Functionally, there is no difference between importing all submodules at once, or only the bits and pieces that you actually need. In fact, from a functional point of view, the former is far more convenient. Potential advantages of more fine-grained submodule imports, however, include improved compilation speeds (fewer dependencies between source files mean more opportunities for parallel compilation) and reduced incremental build times during development (when you alter the interface of a submodule, only files that import that specific submodule need to be rebuilt).

### Module Partitions
If splitting up a module into multiple (sub)modules does not suit you, *module partitions* offer you the means to bring more structure to otherwise overly large, monolithic module files.

### Note
The key difference between submodules and partitions is that submodules can be imported individually by the rest of the application, whereas partitions are only visible within a module. For the rest of the application, a partitioned module still presents itself as one, self-contained module. You can even completely repartition a module without changing anything else in the remainder of the application.

### Module Implementation Partitions
You already know from Ex11_02 that you can divide a module’s implementation across multiple module implementation files. But this is not always enough; sometimes you want to use the same module-local entities within multiple of these source files—preferably without having to declare these implementation details in the shared module interface file. This is what *module implementation partitions* are for.

One example would be the following partition, extracted from our massive `roman` module. It defines the `from_roman(char)` helper previously defined in `from_roman.cpp` (the complete source for this repartitioned `roman` module is available as Ex11_04).

---

8 Module implementation partitions are sometimes also called *internal partitions*. 

378
The module declaration of a module partition file is similar to that of any other module file, except that the name of the partition is added after the name of the module, separated by a colon. In our example the name of the partition is thus `internals`, and clearly it is a partition of the `roman` module. The lack of an export keyword in the module declaration signals that this is a module implementation partition, and that it therefore is not allowed to export any entities. The name `internals` uniquely identifies this specific file.

**Caution** No two module partition files can ever share the same partition name—not even if one of these files contains a module interface partition and the other a module implementation partition. A partition name thus always uniquely identifies one single file.

To then use the `internals` partition and its function in the module implementation file that defines `from_roman(std::string_view)`, you have to import the partition, as shown next.

```
// from_roman.cpp - Implementation of the from_roman() function
module roman;
import :internals;    // Caution: never 'import roman:internals;'

unsigned int from_roman(std::string_view roman)
{
    // Same as before... (uses from_roman(char) function from the :internals partition)
}
```

**Caution** Module partitions can only be imported into other files of the same module, and only using `import :partition_name;`. The alluring syntax `import module_name:partition_name;` is never valid (neither outside the module, where the partition cannot be accessed, nor within a module, where the `module_name` qualification would only be redundant anyway).

Naturally, grouping module-local functionality into implementation partition files only really pays off if these partitions are subsequently reused by multiple other implementation and/or partition files of the same module. If a partition such as `internals` is only used by one other file, it may not be worth the effort.
Module Interface Partitions

Even module interfaces can at times become large enough to warrant a subdivision in multiple parts. For this you can use module interface partitions. The interface of the roman module of Ex11_03, for instance, contains two functions. TWO! Clearly way too much for one interface to hold, so we better split this interface into parts.

Creating a module interface partition is easy enough. Here is a first example in which we export and define the to_roman() function (the complete source is available in Ex11_05).

// roman-to.cppm – Module interface file for the to partition of the roman module
export module roman:to;
import <string>;

export std::string to_roman(unsigned int i)
{
    // Same function body as before...
}

Like in any module partition file, the module declaration of a module interface partition file specifies a unique partition name (to in our case). And like in any module interface file, this declaration begins with the keyword export. Using module interface partitions such as roman-to.cppm you can divide your module into multiple parts, each defining some logical subset of the module’s functionality.

Of course using module interface partitions does not preclude separating the interface from the implementation either. To illustrate that, we will move all definitions of our second interface partition, roman:from, to an implementation file.

// roman-from.cppm – Module interface file for the from partition of the roman module
export module roman:from;
import <string_view>;

export unsigned int from_roman(std::string_view roman);

There’s just one thing to watch out for now—you cannot, at this point, create a module implementation file that starts with module roman:from; While this may seem logical, only one file can be created per partition name. So once you create a module interface partition named from, you can no longer create a module implementation partition with that same name. What you can do, however, is move the partition’s function definitions to a regular, unnamed module implementation file (or, if you want, a module implementation partition with a name that is not from). Because we have it lying around, we also reuse the internals partition from Ex11_04 in our implementation.

// from_roman.cpp – Module implementation file for the from partition of the roman module
module roman;
import :internals;

unsigned int from_roman(std::string_view roman)
{
    // Same as before... (uses from_roman(char) from the internals partition)
}

However you organize your module interface partitions and their implementation, the key is that every interface partition must eventually be exported from the module’s so-called primary module interface file. Remember, to the outside world, a partitioned module still presents itself as a single, self-contained module.
Each module must therefore have exactly one primary module interface file that exports its entire interface, even if that interface is internally partitioned across multiple files. This is the primary module file that completes the roman module of our running example, Ex11_05.

```cpp
// roman.cppm – Primary module interface file for the roman module
export module roman;

export import :to;       // Not: 'export import roman:to';
export import :from;     // Not: 'export import roman:from';
// export import :internals;  /* Error: only interface partitions can be exported */
```

You cannot export a module implementation partition such as :internals; only module interface partitions can (and must) be exported. Outside of the module, a partitioned module can only be imported as a whole (through import roman; for instance); individual partitions can never be imported outside of the module (the syntax import roman:from; as said, is therefore never valid). If you want individual parts to be importable from outside the main module, you should turn them into submodules instead.

**Global Module Fragments**

Modules were only introduced in C++20. That means there is over three decades' worth of nonmodular C++ code out there. All this code is said to be in the global module, an implicit module with no real name. All code that you wrote prior to this chapter resides in the global module. You will learn how nonmodular code was organized into multiple files (into headers and source files, to be exact) in online Appendix A; in this section, we simply show you how to consume (truly) nonmodular code inside module files.

Many if not most C++ header files are said to be importable, meaning you can readily import them as if they were modules. We'll have more to say about what makes a header importable or not in Appendix A; for now it suffices to know that all headers of the C++ Standard Library are importable, except for those headers that originate from the C Standard Library (that is, `<cmath>`, `<cctype>`, and so on; their names are all prefixed with the letter ‘c’). The standard C headers are not necessarily importable. From Chapters 2 and 4, you already know that in order to consume their functionality you should therefore use #include directives instead of import declarations.

In a module file, as a rule, anys and all #include directives should be put in a so-called global module fragment. Here is the typical layout of a module file with a global module fragment:

```cpp
module;            // Start of the global module fragment
#include <cmath>   // Any number of #includes of nonmodular code (no semicolon!)
/*export*/ module myModule; // Start of the module purview
import <string>;            // Any number of imports of modular code
import myOtherModule;

// Module code that consumes entities from <cmath>, <string>, and myOtherModule...
```

A global module fragment always comes before the module declaration of a module file. In fact, other than comments, nothing but a global module fragment may appear before the module declaration. The section of the module file that starts with the module declaration then is formally called the module purview.
The global module fragment itself may only contain preprocessor directives, such as `#include` and `#define`. Nothing else: The global module fragment may not contain any function declarations, no variable definitions, nothing. All regular declarations belong in the module purview. Refer to online Appendix A for more details about preprocessor directives. Until then, we will only occasionally need a global module fragment to `#include` some C headers of the Standard Library.

Note: The `module;` line at the start of a global module fragment may seem redundant. The reason they made it compulsory is to make it easier for compilers and build systems to quickly determine whether a given file is a module file or not (without having to process any of the `#include` directives).

Namespaces

We briefly introduced namespaces in Chapter 1, but there’s a bit more to it than we explained then. With large programs, choosing unique names for all the entities can become difficult. When an application is developed by several programmers working in parallel and/or when it incorporates code from various third-party C++ libraries, using namespaces to prevent name clashes becomes essential.

A namespace is a block that attaches an extra name—the namespace name—to every entity name that is declared within it. The full name of each entity is the namespace name followed by the scope resolution operator, `::`, followed by the basic entity name. Different namespaces can contain entities with the same name, but the entities are differentiated because they are qualified by different namespace names. You are already aware that Standard Library names are declared within the `std` namespace. In this section you will learn how to define namespaces of your own. But before we dive in to that, first a quick word on the so-called global namespace.

The Global Namespace

All the programs that you’ve written so far have used names that you defined in the global namespace. The global namespace applies by default if a namespace hasn’t been defined. All names within the global namespace are just as you declare them, without a namespace name being attached.

To explicitly access names defined in the global namespace, you use the scope resolution operator without a left operand, for example, `::power(2.0, 3)`. This is only really required, though, if there is a more local declaration with the same name that hides that global name.

Note: A name can be hidden by the name of a totally different kind of entity. In the following function body, for instance, the name of the function parameter `power` hides that of the afore-alluded-to `power()` function in the global namespace. To access the latter, you therefore have to explicitly add the scope resolution operator (or, of course, rename the function parameter).

```cpp
double zipADee(int power) // Compute the zip-a-dee of the given power
{
    double doodah = ::power(std::numbers::pi, power);
    // ...
```
With small programs, you can define names within the global namespace without running into any problems. Within larger code bases the potential for name clashes increases, so you should use namespaces to partition your code into logical groupings. That way, each code segment is self-contained from a naming perspective, and name clashes are prevented.

**Defining a Namespace**

A namespace definition has the following form:

```cpp
namespace mySpace
{
    // Code you want to have in the namespace,
    // including function definitions and declarations,
    // global variables, enum types, templates, etc.
}
```

Note that no semicolon is required after the closing brace in a namespace definition. The namespace name here is `mySpace`. The braces enclose the scope for the namespace `mySpace`, and every name declared within the namespace scope has the name `mySpace` attached to it.

---

**Caution** You must not include the `main()` function within a namespace. The runtime environment expects `main()` to be defined in the global namespace.

---

You can extend a namespace scope by adding a second namespace block with the same name. For example, a program file might contain the following:

```cpp
namespace mySpace
{
    // This defines namespace mySpace
    // The initial code in the namespace goes here
}
namespace network
{
    // Code in a new namespace, network
}
namespace mySpace
{
    /* This extends the mySpace namespace
    Code in here can refer to names in the previous
    mySpace namespace block without qualification */
}
```

---

9 Not to be confused with Myspace™, once the largest social networking site in the world. (We checked: Myspace™ was originally stylized as “MySpace” and now as “myspace”; so, yes, given that namespace names are case sensitive, `mySpace` was still available!)
There are two blocks defined as namespace mySpace, separated by a namespace network. The second mySpace block is treated as a continuation of the first, so functions declared within each of the mySpace blocks belong to the same namespace. You can have as many blocks for the same namespace as you want, spread across as many source and module files as you want.

References to names from inside the same namespace do not need to be qualified. For example, names that are defined in the namespace mySpace can be referenced from within mySpace without qualifying them with the namespace name.

The following example illustrates the mechanics of defining and using a namespace:

```cpp
// Ex11_06.cpp
// Defining and using a namespace
import <iostream>
import <numbers>

namespace math
{
    const double sqrt2 { 1.414213562373095 };      // the square root of 2
    auto square(const auto& x) { return x * x; }
    auto pow4(const auto& x) { return square(square(x)); }
}

int main()
{
    std::cout << "math::sqrt2 has the value " << math::sqrt2 << std::endl;
    std::cout << "This should be 0: " << (math::sqrt2 - std::numbers::sqrt2) << std::endl;
    std::cout << "This should be 2: " << math::square(math::sqrt2) << std::endl;
}
```

The namespace math in Ex11_06 contains one constant and two (abbreviated) function templates. You can see that it is perfectly okay to define a constant named sqrt2 in the math namespace even though a variable with the same name exists in the std::numbers namespace (std::numbers is a nested namespace; we discuss nested namespaces in the next section). As long as you use their qualified names, there is no ambiguity.

Note that you did not have to qualify square() with math:: when calling it (twice) from pow4(). The reason for this is that pow4() is defined in the same namespace as square(). This would still work even if pow4() were defined in a different namespace block than square(), and even if that block were defined in a different file. Of course you do have to qualify square() with math:: when calling it from outside the math namespace, such as from the main() function of Ex11_06.cpp (a function in the global namespace).

This program produces the following output:

```
sqrt2 has the value 1.41421
This should be 0: -2.22045e-16
This should be 2: 2
```

This somewhat surprising second line only reaffirms two things: First, that math::sqrt2 and std::numbers::sqrt2 are effectively two different variables with the same base name; and second, that for optimal accuracy you should use predefined constants such as those of the <numbers> module whenever possible (as recommended in Chapter 2 as well). Believe us, we really did not set out for these two constants to be different!
Nested Namespaces

You can define one namespace inside another. The mechanics of this are easiest to understand if we look at a specific context. For instance, suppose you have the following *nested namespaces*:

```cpp
import <vector>;

namespace outer
{
  double min(const std::vector<double>& data) 
  { 
    // Determine the minimum value and return it...
  }

namespace inner
{
  void normalize(std::vector<double>& data)
  {
    const double minValue{ min(data) };   // Calls min() in outer namespace
    // ...
  }
}
```

From within the `inner` namespace, the `normalize()` function can call the `min()` function in the `outer` namespace without qualifying the name. This is because the declaration of `normalize()` in the `inner` namespace is also within the `outer` namespace. To call `min()` or `normalize()` from the global namespace (or any other unrelated namespace), you of course must qualify both names in the usual way:

```cpp
outer::inner::normalize(data);
const double result{ outer::min(data) };
```

To call `outer::inner::normalize()` from within the `outer` namespace, though, it suffices to qualify its name with `inner::`. That is, you can omit the `outer::` qualifier from names in namespaces that are nested inside `outer`. Here is an example:

```cpp
namespace outer
{
  auto getNormalized(const std::vector<double>& data)
  {
    auto copy{ data }; 
    inner::normalize(copy);   // Same as outer::inner::normalize(copy);
    return copy;
  }
}
```

You can add something to a nested namespace directly using the following compact syntax.

```cpp
namespace outer::inner
{
  double average(const std::vector<double>& data) { /* body code... */ }
}
This is equivalent to—yet far more convenient than—explicitly nesting a block for the inner namespace inside a block for the outer namespace.

```cpp
namespace outer
{
    namespace inner
    {
        double average(const std::vector<double>& data) { /* body code... */ }
    }
}
```

### Namespaces and Modules

You typically use a separate namespace within a single program for each collection of code that encompasses a common purpose. Each namespace would represent some logical grouping of types and functions, together with any related global variables. A namespace could also be used to contain a unit of release, such as a library.

If the previous paragraph sounds familiar, it is only because we wrote the exact same passage earlier in this chapter when talking about modules—word for word. And it’s true, if you look past some of the differences—modules are composable units of code stored in separate files, whereas namespaces are hierarchical groupings of names—both language features often serve the exact same purpose: they group related code entities to bring structure, order, and overview in an ever-growing code base. Therefore also this advice:

■ **Tip** While not required, it only stands to reason that aligning the names of your modules and namespaces is often a good idea. It would make perfect sense, for instance, to package the three entities of the math namespace of Ex11_06 in a math module. That way, you have to remember only one name when using a module and its entities.

Even though consistent naming of modules and namespaces may be recommended, the C++ language by no means enforces it. Unlike in other programming languages, namespaces and modules are completely orthogonal in C++. From a single module you can export entities from as many namespaces as you want, and, conversely, entities from the same namespace can be spread across as many modules as you want.

To illustrate this, you could for instance package the math namespace of Ex11_06, not in a math module, but in a squaring module instead (see Ex11_06A).

```cpp
export module squaring;

namespace math
{
    export const double sqrt2 { 1.414213562373095 }; // the square root of 2
    export auto square(const auto& x) { return x * x; }
    export auto pow4(const auto& x) { return square(square(x)); }
}
```

If all entities of a namespace block are to be exported, you can also move the export keyword in front of the namespace keyword. The following is therefore entirely equivalent to our previous squaring module (see Ex11_06B).
export module squaring;

export namespace math         // Exports all nested declarations at once
{
    const double sqrt2 { 1.414213562373095 };      // the square root of 2
    auto square(const auto& x) { return x * x; }
    auto pow4(const auto& x) { return square(square(x)); }
}

Either way, besides having to import the squaring module, nothing changes for the main() program of Ex11_06.

Organizing Larger Namespaces and Modules

Here are some strategies you should consider for organizing larger modules and/or namespaces:

- You can divide entities of the same namespace across multiple modules. This is essentially what the Standard Library did for the std namespace.
- You can assign some or all modules a matching (nested) namespace. The <ranges> module of the Standard Library, for instance, uses the std::ranges namespace (as detailed in Chapter 20).
- You can also divide all entities into hierarchical submodules. Suppose the math namespace contains many more entities than those listed in Ex11_06. Then math.squaring would be a good name for the modules of Ex11_06A and Ex11_06B. Other submodules could then be math.polynomials, math.geometry, and so on. Using the techniques discussed earlier, you could then make it so that by importing math you import all math submodules at once.
- You can combine hierarchical submodules with a parallel hierarchy of nested namespaces (that is, math::squaring, math::polynomials, and so on).

Whichever scheme you choose, our advice is to try to be consistent within your code base. After all, the goal is to bring structure and order in an otherwise chaotic collection of code.

Functions and Namespaces

For a function to exist within a namespace, it is sufficient for the function prototype to appear in a namespace block. Same as with functions in the global namespace, you can then define the function elsewhere—either in the same file or in, for instance, a module implementation file. To define such a function whose prototype was declared earlier, you have two options: either you again enclose the definition in a namespace block, or you use the function’s qualified name to introduce its definition.

An example will make things more clear. We start from this module interface file for yet another math module.

// math.cppm – Module interface file containing declarations and template definitions
export module math;
import <span>;

export namespace math
{
    auto square(const auto& x) { return x * x; }
}
As said earlier, separating a module’s interface from its implementation allows for an quick and easy overview of a module’s content. This particular math module, for example, clearly focuses on the computation of various types of averages.

To export the (abbreviated) `square<>()` template, its definition must be part of the module interface, as explained earlier as well. You could move its definition either to the bottom of the primary interface file or to an interface partition file, but never to a module implementation file.

The definitions of the functions from the `math::averages` namespace, on the other hand, can be moved into a module implementation file, as shown next. Of course you could simply put them at the bottom of the module interface file as well—and there would be nothing wrong with that. For brevity, we have omitted the function bodies in the following outline, but you can find the full code in Ex11_07.

```cpp
// math.cpp – Module implementation file containing function definitions
module math;  // Remember: this implicitly imports the primary module interface...

// Option 1: define in nested namespace block (compact syntax)
namespace math::averages
{
    double arithmetic_mean(const std::span<double>& data) { /* body code... */ }
}

// Option 2: define in nested namespace blocks
namespace math
{
    namespace averages
    {
        double geometric_mean(const std::span<double>& data) { /* body code... */ }
    }
}

// Option 3: define using fully qualified function name
double math::averages::rms(const std::span<double>& data) { /* body code... */ }

// Option 4: define using qualified name in outer namespace block
namespace math
{
    double averages::median(const std::span<double>& data) { /* body code... */ }
}
```

This implementation file illustrates the different options for defining functions that were previously declared in a (nested) namespace. The first two function definitions are simply placed in equivalent nested namespace blocks. But since all functions have been declared already in their respective namespaces in the implicitly imported module interface file, you no longer have to put their definitions...
in namespace blocks if you don’t want to. You can instead tie the definitions to their declarations by qualifying the function names with the correct namespace names in their definitions. This is illustrated by options 3 and 4 in math.cpp.

It goes without saying that mixing all these different options is not considered good style (and certainly not within one file). Instead, you should pick one option (most commonly option 1 or 3 in math.cpp) and then consistently use that style for all your function definitions.

### Using Directives and Declarations

From Chapter 3 you already know that by using a blanket *using directive* you can reference any name from a namespace without qualifying it with the namespace name:

```cpp
using namespace std;
```

However, this risks defeating the purpose of using namespaces in the first place and increases the likelihood of errors because of the accidental use of a name in the std namespace. It is thus often better to use qualified names or add using declarations for the names from another namespace that you are referencing. A *using declaration*, as you also know from Chapter 3, allows you to use a specific name without its namespace qualification. Here are some examples:

```cpp
using std::vector;
using std::max;
```

These declarations introduce the names `vector` and `max` from the namespace `std` into the current scope or namespace. One using declaration may introduce many different entities. For instance, the using directive for `std::max` introduces all possible instantiations and overloads of the various `std::max<>()` templates with a single using declaration.

We have occasionally placed some using declarations and directives already at global scope in some examples. But you can also place them within a namespace, within a function, or even within a statement block. When placed in a namespace, it makes a name from one namespace available within another. Here is a variation of Ex11_06B that uses this technique:

```cpp
// squaring.cpp
module;               // Start of the global module fragment (for #include directives)
#include <cmath>      // For std::sqrt()
export module squaring;
import <numbers>;     // For std::numbers::sqrt2
export namespace math
{
    using std::numbers::sqrt2;
    using std::sqrt;    // Never 'using std::sqrt();' or 'using std::sqrt(double);'!
    auto square(const auto& x) { return x * x; }
    auto pow4(const auto& x) { return square(square(x)); }
}
```

By adding the `using std::numbers::sqrt2;` declaration, you ensure that by importing squaring you can qualify the `sqrt2` constant both as `std::numbers::sqrt2` and `math::sqrt2`. Note that if the entire `math` namespace was not exported already, you would have to use `export using std::numbers::sqrt2;` to make this alias visible from outside the squaring module.
The second using declaration in the exported math namespace similarly brings all overloads of the sqrt() function into this namespace as well. You cannot pick and choose which overloads of a function to import into a namespace; you can only import a particular name (see also the comment in the code). std::sqrt(), for instance, has overloads for double, float, long double, and various integral parameters; due to our single using declaration all these overloads are addressable now as math::sqrt().

If you want to introduce all constants (and variable templates) of the <numbers> module into the math namespace—instead of only the sqrt2 constant like we did originally—you can do so by adding (and exporting) the following using directive.

```cpp
namespace math
{
    export using namespace std::numbers;
}
```

When adding a using directive or declaration at any scope that is not a namespace, analogous aliasing rules apply, but then only within that scope. Take the following function template, for instance, for functions that compute the length of the hypotenuse of a right-angled triangle given the length of the other two sides (using the well-known Pythagorean theorem). Since this hypot() template is defined in the global namespace, its body would by default have to qualify the sqrt and square names with math::.

```cpp
auto hypot(const auto& x, const auto& y)
{
    using namespace math;
    // Or:
    //   using math::square;
    //   using math::sqrt;  /* Same as, of course: using std::sqrt; */
    return sqrt(square(x) + square(y));
}
```

The declaration or directive applies until the end of the block that contains it.

---

**Caution** Never use this naïve implementation of hypot() in practice. Always use the std::hypot() functions of <cmath> instead. The reason is that our naïve solution often is fairly inaccurate (it suffers both from underflow and overflow issues). Because creating efficient and numerically stable mathematical primitives is deceptively hard (many scientific articles have been written on algorithms for hypot() alone), you should always look for predefined primitives first.

---

**Tip** What we explained here for using directives and declarations also applies for type aliases and using enum declarations (both introduced in Chapter 3). They all can be exported from modules and/or brought into a namespace, or applied more locally at function or even statement scopes. You saw an example of a local using enum declaration inside the switch statement in the from_roman() helper function earlier in this chapter.

Personally, we prefer to apply using directives and declarations only sparsely, and always as locally as possible. That way we do not only avoid inadvertent name clashes, but we also clarify to the reader of our code from which namespace each entity name originates.
Namespace Aliases

Long namespace names, often originating from deeply nested namespaces, may be unduly cumbersome to use. Having to attach names such as `my_excessively_very_long_namespace_name_version2::` or `MyCompany::MyModule::MySubModule::MyGrouping::` to every function call, for instance, would be more than a nuisance. To get over this, you can define an alias for a namespace name on a local basis. The general form of the declaration you'd use to define an alias for a namespace name is as follows:

```cpp
namespace alias_name = original_namespace_name;
```

You can then use `alias_name` in place of `original_namespace_name` to access names within the namespace. For example, to define aliases for the namespace names mentioned in the previous paragraph, you could write this:

```cpp
namespace v2 = my_excessively_very_long_namespace_name_version2;
namespace MyGroup = MyCompany::MyModule::MySubmodule::MyGrouping;
```

Now you can call a function within the original namespace with a statement such as this:

```cpp
int fancyNumber{ v2::doFancyComputation(MyGroup::queryUserInput()) };
```

Summary

This chapter discussed capabilities that operate between, within, and across program files. C++ programs typically consist of many files, and the larger the program, the more files you have to contend with. To keep this code ordered and organized, it's vital that you understand modules and namespaces.

The important points from this chapter include the following:

- Modules allow you to organize your source code into logical, self-contained building blocks.
- Only exported declarations are visible within files that import the module. Other declarations in the module interface files are merely reachable, which mostly suffices to use the module. Declarations and definitions that only exist in module implementation files are unreachable outside of the module.
- You can export basically any declaration from a module: function declarations, variable declarations, type declarations, using declarations, using directives, module import declarations, and so on.
- Only changes in a module's interface (the collection of all exported entities) require you to recompile consumers of the module. Function definitions are not part of the module's interface, only their prototypes are.
- Only module interface files can export declarations, but you can move function definitions and any module local entities into module implementation files to keep the interface file itself clear and simple.
- At the beginning of every module file is a module declaration. In module interface files, this declaration has the form `export module name;`. In module implementation files, it has the form `module name;`. In these declarations, `name` is normally the name of the module; only in module partition files this `name` takes the form `module_name:partition_name`.  

391
• Every module has exactly one primary module interface file (starting with `export module module_name;`). Every module interface partition must be exported, directly or indirectly, from the module’s primary interface file.

• Partitions cannot be imported outside of the module. If you want parts of a module to be importable like that, you can turn them into submodules instead.

• The module declaration may be preceded by a global module fragment, containing preprocessor directives. The global module fragment is where you normally mainly add `#include` directives for headers that are not importable as modules (such as the C headers of the Standard Library).

• Namespaces are hierarchical groupings of entity names, designed to avoid name clashes within larger code bases (and especially when integrating various third-party libraries). While namespaces and modules are completely orthogonal in C++, it only stands to reason that aligning the names of your namespaces and modules is a sensible idea.

• `using` directives and `using` declarations risk nullifying the advantages of namespaces, and they should therefore only be used sparsely, and as locally as possible (ideally even at the level of a function body or statement block).

EXERCISES

The following exercises enable you to try what you’ve learned in this chapter. If you get stuck, look back over the chapter for help. If you’re still stuck after that, you can download the solutions from the Apress website (www.apress.com/source-code), but that really should be a last resort.

Exercise 11-1. One of the larger programs you have seen thus far is that of Ex8_18.cpp. Extract all its functions and put them in a single-file `words` module, with all functions part of a `words` namespace. The module should only export those functions that are relevant to the `main()` function, which for the most part should be kept as is. All other functions, and especially the ternary overload of `sort()`, exist solely to support these exported functions.

Exercise 11-2. Retake your solution to Exercise 11-1 and split the module’s interface from its implementation by moving all function definitions to an implementation file. Do not use namespace blocks in the implementation file. As part of the exercise, move all nonexported functions out of the `words` namespace and into the global namespace (and then figure out how to still invoke the ternary `sort()` function from the exported unary `sort()` function).

Exercise 11-3. Split the insanely large `words` module of Exercise 11-2 into two aptly named submodules: one submodule containing all `sorting` functionality, and one containing any remaining utilities (or `utils`, in programmer’s speak). Also put your functions in nested namespaces whose names mirror those of the submodules.

Exercise 11-4. Start again from the solution of Exercise 11-2 and move `swap()` and `max_word_length()` into an `internals` module implementation partition.
Exercise 11-5. Go back to the solution of Exercise 11-1, only this time instead of creating implementation files as in Exercise 11-2, create multiple interface partition files, each still containing their function definitions. One partition again contains all sorting functionality, the other the remaining utilities (utilis). Also recycle the internals partition of Exercise 11-4.

Exercise 11-6. Start once again from Exercise 11-2. Because the length of words is exorbitant (five letters, oh my!), make sure that all names in this namespace can also be qualified with words (four letters, already much better!) and w (one letter, perfect!). Use two different techniques for this. (For the record, neither cryptic acronyms such as w̓rds nor one-letter identifiers such as w have their place in production-quality code: clarity always trumps compactness!)