APPENDIX A

Preprocessing, ODR, Linkage, and Header Files

In this appendix we discuss some additional aspects of managing source code in multiple files—aspects that mostly become relevant when working with non-modular code. From Chapter 11, you already know how to organize C++ entities of larger programs into composable, self-contained modules. But modules were only introduced in C++20. Prior to C++20, programs were organized in header files and source files. In this appendix, we discuss how non-modular program files interact and how you manage and control their contents.

Including header files is significantly more intricate and low level than importing module files. To understand how you properly divide C++ declarations and definitions in headers and source files, you first need to know the one definition rule, as well as the different types of linkage a name can have in C++. And because the inclusion of header files is both performed and controlled using low-level preprocessing directives, we must first introduce preprocessing as well.

In this appendix, you'll learn

- What a translation unit is
- What preprocessing is and how to use the preprocessing directives to manage code
- How you can define macros, and why C++ typically offers better alternatives
- How to pick and choose what code to compile depending on the active compiler, target platform, or compiler configuration
- The debugging help you can get from the `assert()` macro of the Standard Library
- That ODR is short for One Definition Rule, and why it is especially important to understand these fundamental rules when managing non-modular code
- What linkage is and when it matters
- How you organize declarations and definitions in header and source files in non-modular C++
- The advantages of C++20's modules over legacy header and source files

Translation: From Source to Executable

Creating an executable from a set of C++ source code files is basically a three-phase process[1]: *preprocessing*, *compilation*, and *linking* (see also Chapter 1).

First, the *preprocessor* prepares and modifies each source file according to instructions that you specify by *preprocessing directives*. It recursively analyzes, executes, and then removes all these directives from the source code. The output of the preprocessor is called a *translation unit*, and it consists purely of C++ declarations and statements.

Next, the *compiler* processes each translation unit independently to generate an object file. An object file contains the machine code for the definitions it encountered in the translation unit, as well as references to so-called *external entities*—entities that were used by, but not defined in, the translation unit. Examples of external entities include entities that are defined and exported by an imported module (see Chapter 11). Later in this appendix, you will encounter other examples of external entities as well.

The *linker* combines all object files of a program to produce the final executable. It establishes all necessary connections between all object files for all source files. If an object file contains references to an external entity that is not found in any of the other object files, no executable will result, and there will be one or more error messages from the linker.

The combined process of preprocessing source files, compiling translation units, and linking object files is referred to as *translation*.

■ **Note** For your convenience, the preprocessing and compilation stages are typically performed by one and the same executable (often referred to as "the compiler"), but most compiler systems allow you to invoke the preprocessor separately as well, should you choose to.

Even though the introduction of modules in C++20 (seen in Chapter 11) has had a significant impact on how we translate source code into executables today, the overall three-stage process has remained more or less the same. All source files—including module files—are still preprocessed into translation units and subsequently compiled into binary object files, which are then ultimately all linked together to form an executable. The biggest difference is that with modules the preprocessor no longer textually copy-pastes the source of the entire module interface into each translation unit that imports it. Instead, as explained also in Chapter 11, the compiler relies on a precompiled binary of the module's interface to process any translation unit that consumes it. We'll have more to say about the differences between including headers and importing modules later in this appendix.

Before we can properly explain these differences, though, we first need to explain how header inclusion works exactly. And for that, you need to understand how preprocessing works, and how you can control it using the different preprocessing directives.

Preprocessing Directives

All preprocessing directives begin with the symbol #, so they are easy to distinguish from C++ language declarations. Table A-1 shows the complete set.

[1] Formally, translation actually consists of *nine* distinct phases, not three. Before preprocessing, for instance, there are three more stages: roughly, first all Unicode characters are escaped, then all code comments are removed, and finally lines split using \ are glued together (you will see an example of using \ later in this appendix). There is little point in knowing all nine formal phases of translation, though, which is why we simplify matters here and compress them into three informal ones.

Table A-1. *Preprocessing Directives*

Directive	Description
`#define`	Defines a macro.
`#undef`	Deletes a macro previously defined using `#define`.
`#if`	Start of a conditional preprocessing block.
`#else`	`else` for `#if`. Can only be used once after each `#if`, `#ifdef`, or `#ifndef`.
`#elif`	Equivalent to `#else #if`, except that it does not necessitate an extra `#endif`. An `#if` directive can be followed by any number of `#elif` directives. This sequence of `#elif` directives, then, may be concluded by at most one single `#else` directive, before the entire conditional preprocessing block is terminated with an `#endif` directive.
`#ifdef`	Equivalent to `#if defined`. Start of a conditional preprocessing block whose `if` branch is processed if a macro with the given identifier is defined.
`#ifndef`	Equivalent to `#if !defined`. Start of a conditional preprocessing block whose `if` branch is processed if no macro with the given identifier is defined.
`#endif`	Signals the end of a conditional preprocessing block.
`#error`	Outputs a compile-time error message and stops the compilation. Typically part of a conditional preprocessing block (for obvious reasons).
`#line`	Redefines the current line number. Optionally changes the filename as well.
`#include`	Injects the entire contents of a given file into the current translation unit.
`#pragma`	Offers vendor-specific features. Example uses of the `#pragma` directive include the suppression of compiler warnings, the configuration of compiler optimizations, and so on. You can consult your compiler documentation for a list of supported pragmas and their syntax.

Preprocessing is a low-level mechanism that originates from C, and is becoming less and less relevant with current C++. The capabilities of C++ often provide much more effective and safer ways of achieving the same result as the preprocessing directives of Table A-1. Modules, for instance, provide a far more satisfying way of organizing your code in multiple, self-contained files than you can achieve using `#include` directives. And constants and function templates are far safer and more powerful than the macros you define with the `#define` directive. We will signal these and other alternatives offered by C++ throughout this appendix.

Most preprocessing directives nonetheless do still have their uses, even in modern C++, albeit only sporadically. For instance, when your code has to be compilable with different compilers, or if you target multiple operating systems, the occasional conditional preprocessing block tends to be unavoidable. And of course you will still encounter preprocessing directives in virtually any legacy header or source file. So you for sure still need at least a basic notion of what the different preprocessing directives do.

We begin by explaining how you define macros using the `#define` directive.

Defining Preprocessor Macros

A `#define` directive specifies a so-called *macro*. A macro is a text rewrite rule that instructs the preprocessor which text replacements to apply to the source code prior to handing it over to the compiler. We distinguish two types of macros: *object-like macros* and *function-like macros*.

Defining Object-Like Macros

The simplest form of the #define preprocessing directive is the following:

```
#define IDENTIFIER sequence of characters
```

This so-called *object-like* macro effectively defines IDENTIFIER as an alias for sequence of characters. IDENTIFIER must conform to the usual definition of an identifier in C++; that is, any sequence of letters and digits, the first of which is a letter, and where the underline character counts as a letter. A macro identifier does not have to be in all caps, although this is certainly a widely accepted convention. sequence of characters can be any sequence of characters, including an empty sequence or a sequence that contains whitespace. In fact, object-like macros with empty replacement sequences are relatively common; we discuss this option further in a subsection shortly.

The most obvious use for #define, surely, is to specify some identifier that is to be replaced in the source code by a non-empty substitute string during preprocessing. Here's how you could define PI to be an alias for a sequence of characters that represents a numerical value:

```
#define PI 3.1415
```

Even though this PI macro *looks* like a variable, it has nothing to do with one. PI is a symbol, or *token*, that is to be exchanged for the specified sequence of characters by the preprocessor before the code is compiled. To the preprocessor, 3.1415 is not a numerical value. It is merely text. A string of characters. No validation at all takes place during preprocessing on whether this sequence forms a valid number. If you accidentally wrote 3,1415, or 3. 1415, or 3.14*!5 as the replacement sequence, the substitution would still occur and almost certainly result in invalid C++ code. The preprocessor blindly replaces text, without any safety checks on the character sequence on its right size.

Moreover, if your code contains any C++ entity (a variable, function, type, and so on) named PI, the preprocessor will blindly replace all occurrences of this identifier with 3.1415. Suppose that you didn't know about the PI macro, and you wrote the following otherwise correct C++ statement.

```
const double PI = std::numbers::pi;
```

Then all the preprocessor sees is a token PI, which it has been instructed to replace by the character sequence 3.1415. The preprocessor neither knows nor cares that this token occurs on the left of a C++ assignment. It simply replaces tokens with text. And, so, this is what it hands over to the compiler:

```
const double 3.1415 = std::numbers::pi;
```

Similarly, if the macro was called pi instead of PI, the preprocessor would just as happily rewrite your code into this equally nonsensical statement.

```
const double PI = std::numbers::3.1415;
```

In either case, the compiler will produce a series of cryptic error messages. And, believe us, if you truly have no knowledge of the culprit macros, such errors can be particularly baffling. We will discuss an all too realistic example of such a scenario soon, in the section on function-like macros.

While the #define directive is often used to define *symbolic constants* such as PI in C, it should be clear by now why you should never, ever do this in C++. It is much better and safer to define a constant variable, like this (and even better and safer, of course, to use predefined constants such as those of the standard <numbers> module we introduced in Chapter 2, but that's besides the point):

```
const double PI { 3.14159265358979323846 };
```

■ **Caution** Never use a macro in C++ when a `const` variable (or variable template) will do. Using a `#define` directive has at least three major disadvantages: there's no type checking support, it doesn't respect scope, and the identifier name cannot be bound within a namespace.

Even though the preprocessor will happily rewrite perfectly correct C++ into complete and utter gibberish if you're not careful, it is actually not entirely agnostic about C++'s syntax. It replaces identifiers only when they are tokens; never if they are part of a longer identifier, a string literal, or a code comment. That is, with our earlier macro definition, the preprocessor will not replace the `PI` character sequences in C++ identifiers such as `ALPHA_PI_ALPHA`, in string literals such as `"What famous scientist is born on PI day (March 14)?"`[2], or in C++ code comments such as `/* PI = icRT */` or `// The millionth digit of PI is 5`.

Here's an example program that uses some object-like macro definitions:

```
// ExA_01.cpp
// Defining object-like macros
import <iostream>;

#define POINTER_SIZE sizeof(int*) * BITS_PER_BYTE
#define BITS_PER_BYTE 8

int main()
{
  std::cout << POINTER_SIZE << std::endl;   // 32 or 64, normally
}

// The next macro (while providing a perfectly valid definition for "main")
// is never applied, because it is not defined yet when the preprocessor
// processes the line "int main()".
#define main Of chief or leading importance; prime, principal.
```

With these macros, the preprocessor first replaces any occurrence of `POINTER_SIZE` with the character sequence `sizeof(int*) * BITS_PER_BYTE`, which it then further rewrites to `sizeof(int*) * 8`. That is, after expanding a macro, the preprocessor processes the resulting character sequence again from the start, replacing more tokens where needed. This process continues until there are no more tokens left to replace.

As you know from Chapter 2, the `sizeof` operator evaluates to the number of bytes that is used to represent a type or variable. We arbitrarily choose the type `int*` in `ExA_01`, but any other pointer type would do as well. For a 32-bit program, any pointer occupies 4 bytes, or 32 bits; for a 64-bit program, any pointer occupies 8 bytes, or 64 bits. The output of `ExA_01` therefore typically reads either 32 or 64.

The preprocessor processes a file from top to bottom, executing directives and performing text replacements as it goes along. This is why at the moment that the preprocessor reaches the line with `int main()`, there is no macro defined yet with identifier `main`. This macro only becomes defined a few lines later, and is therefore never applied.

[2] The answer is Albert Einstein (Albert Einstein was born on Pi Day in 1879). Did you also know that Stephen Hawking passed away on Pi Day in 2018? And that he was born 8 January 1942, precisely 300 years after Galileo Galilei died on that exact same day—8 January 1642? Coincidence? We believe so…

Note also that we purposely defined the BITS_PER_BYTE macro *after* the POINTER_SIZE macro to illustrate the order in which text replacements occur. That is, the preprocessor does not yet process the right side of a macro definition when executing a #define directive; it only does so after each macro expansion.

Defining Empty Macros

There's no restriction on the sequence of characters that is to replace the identifier. It can even be absent, in which case the identifier exists but with no predefined substitution string—the substitution string is empty. If you don't specify a substitution string for an identifier, then occurrences of the identifier in the code will be replaced by an empty string; in other words, the identifier will be removed. Here's an example:

```
#define VALUE
```

The effect is that all occurrences of VALUE that follow the directive will be removed.

More often than not, however, you do not define empty object-like macros to perform text replacement. In our example, the same directive also defines VALUE as an identifier whose existence can be tested by other directives, and then used for conditional compilation. The single most common use of this technique in standard C++ is probably the management of header files, as you'll see near the end of this appendix. Another typical use is conditional compilation based on macros set by the compiler or build system. Such macros may, for instance, signal properties of the target platform (such as the operating system, address size, and so on), or certain compiler settings (toggling of debug builds), and so on. You will see examples of conditional compilation shortly, after a brief introduction to function-like macros.

Defining Function-Like Macros

Besides object-like macros, you can also define *function-like* text-replacement macros with #define directives. Here's an example:

```
#define MAX(A, B) A >= B ? A : B
```

While this macro looks an awful lot like a function, it most certainly isn't one. There are no argument types, nor is there a return value. A macro is not something that is called, nor does its right side necessarily specify statements to be executed at runtime. Our sample macro simply instructs the preprocessor to replace all occurrences of MAX(anything1, anything2) in the source code with the character sequence that appears in the second half of the #define directive. During this replacement process, all occurrences of the macro's parameter identifier A in A >= B ? A : B are of course replaced by anything1, and all occurrences of B are replaced with anything2. The preprocessor makes no attempt at interpreting or evaluating the anything1 and anything2 character sequences; all it does again is blind text replacement. Suppose, for instance, that your code contains this statement:

```
std::cout << MAX(expensive_computation(), 0) << std::endl;
```

Then the preprocessor expands it to the following source code before handing it over to the compiler:

```
std::cout << expensive_computation() >= 0 ? expensive_computation() : 0 << std::endl;
```

This example exposes two problems:

- The resulting code will not compile. If you use the ternary operator `?:` together with the streaming operator `<<`, the C++ operator precedence rules (seen in Chapter 3) tell us that the expression with the ternary operator should be between parentheses. A better definition of our `MAX()` macro would therefore be the following:

  ```
  #define MAX(A, B) (A >= B ? A : B)
  ```

  Even better would be to add parentheses around all occurrences of `A` and `B` to avoid similar operator precedence problems there as well:

  ```
  #define MAX(A, B) ((A) >= (B) ? (A) : (B))
  ```

  The bottom line is, though, that making sure that even the simplest of macros always expands to valid code can be surprisingly hard.

- The `expensive_computation()` function is called up to two times. If a macro parameter such as `A` appears more than once in the replacement, the preprocessor will blindly copy the macro arguments more than once. This undesired behavior with macros is not only harder to avoid, but also harder to discover (given that it compiles without error).

These are just two of the common pitfalls with macro definitions. We therefore recommend you create function-like macros only if you have a good reason for doing so. While some advanced scenarios do call for macros, C++ mostly offers superior alternatives. Macros are popular among C programmers because they allow the creation of function-like constructs that work with any parameter type. But you already know that C++ offers a much better solution for this: function templates. After Chapter 10, it should be a breeze for you to define a C++ function template that replaces the C-style `MAX()` macro. C++ function templates inherently avoid both of the shortcomings of macro definitions we listed earlier.

■ **Caution** Never use preprocessor macros where a function template will do. Instead, you should always use either (overloaded) C++ functions or function templates. Function templates are far superior to preprocessor macros for defining blueprints of functions that work for any argument type. C++ functions are type safe, respect scope, can be overloaded, their names can be bound to a namespace, and they are much easier to debug than macros. Function arguments, moreover, are never evaluated more than once.

■ **Note** Because of all the aforementioned downsides of macros, macros cannot even be exported from C++20 modules anymore. Another solid reason to always define proper variables, functions, and templates instead of macros in modern C++.

Preprocessor Operators

For completeness, Table A-2 lists the two operators you can apply to the parameters of a function-like macro.

Table A-2. *Preprocessor Operators*

#	The so-called stringification operator. Turns the argument in a string literal containing its value (by surrounding it with double quotes and adding the necessary character escape sequences).
##	The concatenation operator. Concatenates (pastes together, similar to what the + operator does for the values of two `std::strings`) the values of two identifiers.

The following toy program illustrates how you might use these operators:

```
// ExA_02.cpp
// Working with preprocessor operators
import <iostream>;

#define DEFINE_PRINT_FUNCTION(NAME, COUNT, VALUE) \
  void NAME##COUNT() { std::cout << #VALUE << std::endl; }

DEFINE_PRINT_FUNCTION(fun, 123, Test 1 "2" 3)

int main()
{
  fun123();
}
```

Before we get to the use of both preprocessor operators, `ExA_02.cpp` shows one additional thing: macro definitions are not really allowed to span multiple lines. By default, the preprocessor simply replaces any occurrences of the macro's identifier (possibly taking a number of arguments) with all the characters it finds on the same line to the right of the identifier. However, it is not always practical to fit the entire definition on one single line. The preprocessor therefore allows you to add line breaks, as long as they are immediately preceded by a backslash character. All such escaped line breaks are discarded from the substitution. That is, the preprocessor first concatenates the entire macro definition back into one single line (in fact, it does so even outside of a macro definition).

■ **Note** In `ExA_02.cpp`, we added the line break before the right side of the macro definition, which is probably the most natural thing to do. But since the preprocessor always just stitches any sliced lines back together, without interpreting the characters, such escaped line breaks can really appear anywhere you want. Not that this is in any way recommended, but this means you *could* in extremis even write the following, perfectly valid C++ program (equivalent to `ExA_02.cpp`):

```
import <iostream>;

#define DEFINE_PRINT_FUNCT\
ION(NAME, COUNT, VALUE) vo\
id NAME##COUNT() { std::co\
ut << #VALUE << std::endl; }
```

```
DEFINE_PRINT_FUNCTION(fun, 123, Test 1 "2" 3)

\
i\
nt\
 ma\
in()\
{fun1\
23();}\
```

Mind you, if you do splice identifiers like this, for whatever crazy reason, take care not to add unwanted whitespace characters at the beginning of the next line, as these are not discarded by the preprocessor when it puts the pieces back together.

Enough about multi-line macros and line breaks, though; it's time to get back to the topic at hand: preprocessor operators. The macro definition in `ExA_02` uses both ## and #:

```
#define DEFINE_PRINT_FUNCTION(NAME, COUNT, VALUE) \
  void NAME##COUNT() { std::cout << #VALUE << std::endl; }
```

With this definition, the line `DEFINE_PRINT_FUNCTION(fun, 123, Test 1 "2" 3)` in `ExA_02.cpp` expands to the following function definition:

```
  void fun123() { std::cout << "Test 1 \"2\" 3" << std::endl; }
```

The tokens `fun` and `123` are concatentated with ##. Without the ## operator, you'd have the choice between either `NAMECOUNT` or `NAME COUNT`. In the former, the preprocessor would not recognize `NAME` or `COUNT`; with the latter, our example macro would expand to `fun 123`, which is not a valid function name (C++ identifiers may not contain spaces).

Without the # operator, then, your best attempt at turning `VALUE`'s value into a string literal would be `"VALUE"`, but that does not work for two reasons. First, `"VALUE"` would simply result in the string literal `"VALUE"`, as the preprocessor again does not replace occurrences of macro parameter identifiers inside literals, identifiers, or code comments. And even if we ignore that minor inconvenience, there would be no way for you to inject the required escape characters (\) in front of characters such as double quotes in more complex cases such as our deviously crafted `Test 1 "2" 3`.

Because the preprocessor runs first, the definition of the `fun123()` function will be present by the time the C++ compiler gets its hands on the code. This is why you can call `fun123()` in the program's `main()` function, where it produces the following result:

```
Test 1 "2" 3
```

`ExA_02` thus illustrates that you are by no means limited to specifying only variable- or function-like constructs with macros. That is, you can also use macros to generate entire function definitions, or even entire class definitions if you want to.

Undefining Macros

You may want the macro resulting from a #define directive to be applied only to *part* of a program file. Putting the macro definition in a scope delimited by curly braces ({ and }), however, does not work. Preprocessing macros do not respect scoping, as illustrated by the following example.

```
{
   #define PI 3.141592653589793238462643383279502884
   // ...
}
// All occurrences of PI in code from this point are still replaced!
```

What you can do, however, is nullify a definition for an identifier using the #undef directive. You can negate a previously defined macro with this directive:

```
#undef IDENTIFIER
```

IDENTIFIER is no longer defined following this directive, so no substitutions for IDENTIFIER occur below that line anymore. The following code fragment illustrates this:

```
#define PI 3.141592653589793238462643383279502884
// ...
#undef PI
// PI is no longer defined from here on so no substitutions occur.
// Any references to PI will be left in the code below.
```

Between the #define and #undef directives, preprocessing replaces appropriate occurrences of PI in the code with 3.141592653589793238462643383279502884. Elsewhere, occurrences of PI are left as they are. The #undef directive also works for function-like macros. Here's an example:

```
#undef MAX
```

■ **Tip** Functions of the native API of the Windows operating system are made available through headers such as <windows.h>. One well-known annoyance with this and other Windows headers is that, by default, they define min() and max() preprocessing macros analogous to our MAX() macro earlier (the #include directive and how it may introduce macro definitions into a source file is discussed later in this appendix). For starters, these macros of course suffer from the same issue as any function-like macro: unlike functions, their arguments are potentially evaluated twice, which may inadvertently lead to poor performance. At least as annoying, though, is that the min() and max() macros that are injected by the Windows headers invalidate otherwise perfectly correct C++ expressions such as std::max(0, my_val) because they instruct the preprocessor to rewrite these expressions into utter nonsense like std::((0) > (my_val) ? (0) : (my_val)).

One solution is to undefine both macros after including one of these Windows headers as follows:

```
#include <windows.h>
#undef min
#undef max
```

Another solution[3] is to define the NOMINMAX macro prior to the #include directive, like so:

```
#define NOMINMAX
#include <windows.h>
```

This causes the Windows headers to no longer define the min() and max() macros in the first place, using the conditional compilation techniques we discuss in the next section.

Conditional Compilation

The logical #if works in essentially the same way as an if statement in C++. Among other things, this allows conditional inclusion of code and/or further preprocessing directives (including macro definitions!) in a file, depending on whether preprocessing identifiers have been defined or based on identifiers having specific values. This is particularly useful when you want to maintain one set of code for an application that may be compiled and linked to run in different hardware or operating system environments. You can define preprocessing identifiers that specify the environment for which the code is to be compiled and select either code or #include or #define directives accordingly.

Testing Whether Identifiers Are Defined

You can use the #if defined directive to test whether a given identifier has been defined and include code or not in the file depending on the result:

```
#if defined MY_IDENTIFIER
  // The code here will be placed in the source file if MY_IDENTIFIER has been defined.
  // Otherwise it will be omitted.
#endif
```

All the lines following #if defined up to the #endif directive will be kept in the file if the identifier, MY_IDENTIFIER, has been defined previously and will be omitted if it has not. The #endif directive marks the end of the text that is controlled by the #if defined directive. You can use the abbreviated form, #ifdef, if you prefer:

```
#ifdef MY_IDENTIFIER
  // The code here will be placed in the source file if MY_IDENTIFIER has been defined.
  // Otherwise it will be omitted.
#endif
```

[3] A third, in this case less elegant, solution is to resort to expressions such as (std::max)(0, my_value). By adding an extra pair of parentheses around std::max, you turn this qualified name into a single token in the eyes of the preprocessor, which prevents the unwanted macro expansion.

Similarly, you can use the `#if !defined` or its equivalent `#ifndef` to test for an identifier not having been defined:

```
#if !defined MY_IDENTIFIER
  // The code down to #endif will be placed in the source file
  // if MY_IDENTIFIER has NOT been defined. Otherwise, the code will be omitted.
#endif
```

Suppose you put the following code in your program file:

```
// Code that sets up the array data[]...

#ifdef CALC_AVERAGE
  double average {};
  for (size_t i {}; i < std::size(data); ++i)
    average += data[i];
  average /= std::size(data);
  std::cout << "Average of data array is " << average << std::endl;
#endif

// rest of the program...
```

If the `CALC_AVERAGE` identifier has been defined by a previous preprocessing directive, the code between the `#if` and `#endif` directives is compiled as part of the program. If `CALC_AVERAGE` has not been defined, the code won't be included.

Because the compiler does not even get to see code that is part of a conditional block for which the condition is not met, it may even contain statements that it would not know how to compile at all. This is why such techniques are often used to conditionally compile code based on macros that are only defined for a particular compiler, or when targeting a particular operating system, processor type, and so on. The idea is that code in these blocks then only compiles and/or works when these macros are effectively defined.

Besides testing whether a given macro is defined, you can of course more generally use the `#if` directive to test whether any constant expression is `true`. Let's explore that a bit further.

Testing for Specific Identifier Values

The general form of the `#if` directive is as follows:

```
#if constant_expression
```

The `constant_expression` must be an integral constant expression that does not contain casts. All arithmetic operations are executed with the values treated as type `long` or `unsigned long`, though Boolean operators (`||`, `&&`, and `!`) are definitely supported as well. If the value of `constant_expression` evaluates to nonzero, then lines following the `#if` down to the `#endif` will be included in the code to be compiled. The most common application of this uses simple comparisons to check for a particular identifier value. For example, you might have the following sequence of statements:

```
#if ADDR == 64
  // Code taking advantage of 64-bit addressing...
#endif
```

The statements between the #if directive and #endif are included in the program only if the identifier ADDR has been defined as 64 in a previous #define directive.

■ **Tip** There is no cross-platform macro identifier to detect whether the current target platform uses 64-bit addressing. Most compilers, however, do offer some platform-specific macro that it will define for you whenever it's targeting a 64-bit platform. A concrete test that should work for the Visual C++, GCC, and Clang compilers, for instance, would look something like this:

```
#if _WIN64 || __x86_64__ || __ppc64__
  // Code taking advantage of 64-bit addressing...
#endif
```

Consult your compiler documentation for these and other predefined macro identifiers.

Multiple-Choice Code Selection

The #else directive works in the same way as the C++ else statement, in that it identifies a sequence of lines to be included in the file if the #if condition fails. This provides a choice of two blocks, one of which will be incorporated into the final source. Here's an example:

```
#if ADDR == 64
  std::cout << "Standard 64-bit addressing version." << std::endl;
  // Code taking advantage of 64-bit addressing...
#else
  std::cout << "Legacy 32-bit addressing version." << std::endl;
  // Code for older 32-bit processors...
#endif
```

One or the other of these sequences of statements will be included in the file, depending on whether ADDR has been defined as 64.

There is a special form of #if for multiple-choice selections. This is the #elif directive, which has the following general form:

```
#elif constant_expression
```

The nice thing about the #elif directive is that, unlike with the nearly equivalent #else #if construct, you still only need one single #endif to end the conditional compilation block. Here is an example of how you might use this:

```
#if LANGUAGE == ENGLISH
  #define Greeting "Good Morning."
#elif LANGUAGE == GERMAN
  #define Greeting "Guten Tag."
#elif LANGUAGE == FRENCH
  #define Greeting "Bonjour."
```

```
#else
  #define Greeting "Ola."
#endif
  std::cout << Greeting << std::endl;
```

With this sequence of directives, the output statement will display one of a number of different greetings, depending on the value assigned to LANGUAGE in a previous #define directive.

Caution Any undefined identifiers that appear after the conditional directives #if and #elif are replaced with the number 0. This implies that, should LANGUAGE for instance not be defined in the earlier example, it may still compare equal to ENGLISH, GERMAN, or FRENCH should any of these three either be undefined as well, or explicitly defined to be zero.

Another possible use is to include different code depending on an identifier that represents a version number:

```
#if VERSION == 3
  // Code for version 3 here...
#elif VERSION == 2
  // Code for version 2 here...
#else
  // Code for original version 1 here...
#endif
```

This allows you to maintain a single source file that compiles to produce different versions of the program depending on how VERSION has been set in a #define directive.

Tip Your compiler likely allows you to specify the value of preprocessing identifiers by passing a command-line argument to the compiler (if you're using a graphical IDE, there should be a corresponding properties dialog somewhere). That way, you can compile different versions or configurations of the same program without changing any code.

Testing for Available Headers

Each version of the Standard Library provides a multitude of new header files. These new features and functionalities allow you to write code that would otherwise have taken you a lot more effort, or that would otherwise have been far less performant or robust. On the one hand, you therefore normally always want to use the best and latest that C++ has to offer. On the other hand, though, your code is sometimes supposed to compile and run correctly with multiple compilers—either multiple versions of the same compiler or different compilers for different target platforms. This sometimes requires a way for you to test, at compile time, which headers the current compiler can access to enable or disable different versions of your code.

The `__has_include()` macro can be used to check for the availability of any header file, be it part of either the Standard Library or some other external library, or one of your own headers. Here is an example:

```
#if __has_include(<execution>)
  #include <algorithm> // Or 'import <algorithm>;' and 'import <execution>;' (see later)
  #include <execution>

  // ... Definitions that use std::for_each(std::execution::par, ...)
  // (Note: Chapter 20 explains how to use parallel execution policies)
#elif __has_include(<tbb/parallel_for.h>)
  #include <tbb/parallel_for.h>

  // ... Definitions that use tbb::parallel_for(...) from Intel's TBB library
#else
  #error("Support for parallel loops is required for acceptable performance")
  // ... Alternative is to provide definitions that use slower, sequential for loops
#endif
```

We're sure that you can figure out how this works from this example outline alone, at least as far as the conditional compilation is concerned (the `#include` directive is discussed in more detail later).

Testing for Available Language Features

With the `__has_include()` macro from the previous section, you can check whether some Standard Library header exists, but it does not tell you anything about the content of that header. New versions of C++ regularly add functionality to existing headers as well. C++20, for instance, added the `unseq` execution policy object to C++17's `<execution>` header. The condition `__has_include(<execution>)` from before tells you nothing about whether this new policy is supported. Other examples of new C++20 functionality in existing headers include the extension of `std::make_shared<>()` (see Chapter 6) to support the creation of shared arrays, and the addition of the `std::lerp<>()` and `midpoint<>()` linear interpolation[4] functions to `<cmath>` (see Chapter 2). How do you know whether your code can take advantage of these features, or that it should instead fall back to potentially less effective alternatives?

As of C++20, you can test whether your compiler supports a particular Standard Library feature using the macros defined in the `<version>` header. Here is how you could test for the different features we mentioned in the previous paragraph. For brevity, we omit the actual conditional blocks and `#endif` directives.

```
#if __cpp_lib_execution              // The <execution> header exists (C++17)
#if __cpp_lib_execution >= 201902L   // <execution> defines unseq policy (C++20)

#if __cpp_lib_shared_ptr_arrays      // std::shared_ptr<> supports arrays (C++17)
#if __cpp_lib_shared_ptr_arrays >= 201707L // std::make_shared<>() supports arrays (C++20)

#if __cpp_lib_interpolate       // std::lerp<>() and midpoint<>() added to <cmath> (C++20)
#ifdef __cpp_lib_interpolate         // Equivalent to previous
#if __cpp_lib_interpolate >= 201902L // Equivalent to previous (201902L is initial value)
```

[4] "lerp" is a contraction of "linear interpolation" commonly used in computing and mathematical jargon.

Once a Standard Library implementation fully supports a new library feature, the corresponding *feature testing macro* is either defined or updated. From then on, that macro rewrites to an integer number that denotes the month in which the corresponding feature was officially added to the C++ standard. This number increases whenever that feature is updated significantly. For instance, because the specification for the `<execution>` header was first added to the C++ standard in March 2016, `__cpp_lib_execution` will always be defined as a value that is higher or equal to `201603L` by a Standard Library that provides this header. If your implementation added support for the `unseq` policy, though, the macro will have been redefined to expand to the value `201902L`, which of course denotes February 2019, the month in which this policy was added to the draft of the C++20 standard.

The `<version>` header defines over 150 *library feature test macros* like this, so you'll forgive us that we don't cover them all. You can consult a Standard Library reference should you ever need to know what macros are available, and what their possible values are.

Of course the Standard Library is not the only thing that evolves, so does the C++ language itself. Analogous macros therefore exist to test whether your compiler already supports a given language feature. `__cpp_lambdas`, for instance, is defined if your compiler supports lambda expressions (see Chapter 18), `__cpp_concepts` is defined if it supports concepts (Chapter 21), and so on. These *language feature test macros* (there's about 60 of them at the moment) are implicitly defined in every translation unit, without including any header file or importing any module.

Most feature test macros are object-like. The only exception is `__has_cpp_attribute()`. You can use this function-like macro to test whether your compiler supports a given C++ attribute, such as the `[[fallthrough]]` attribute we discussed in Chapter 4. Here is how you could define a `FALLTHROUGH` macro that only expands to an actual fallthrough statement, `[[fallthrough]];`, if your compiler is known to support it; otherwise the (still self-documenting) identifier `FALLTHROUGH` is simply removed by the preprocessor.

```
#ifdef __has_cpp_attribute
  #if __has_cpp_attribute(fallthrough)
    #define FALLTHROUGH [[fallthrough]];
  #else
    #define FALLTHROUGH
  #endif
#else
  #define FALLTHROUGH
#endif
```

Note that, to be fully compatible with older compilers, we first have to check whether the `__has_cpp_attribute` macro itself is defined. Without this macro, there is no reliable, standardized means to check whether your compiler supports the `[[fallthrough]]` attribute.

■ **Note** Even though the feature test macros have only been added to the official standard in C++20, several major compilers have been supporting these macros since about 2013. Unfortunately, though, not all major compilers followed suit on this at the time, so to be fully compatible with all legacy compilers, you may still have to rely on non-standard means for the time being.

Standard Preprocessing Macros

There are several more standard predefined preprocessing macros besides the feature test macros discussed in the previous section. Table A-3 list some useful examples.

Table A-3. *Predefined Preprocessing Macros*

Macro	Description
`__DATE__`	The date when the source file was preprocessed as a character string literal in the form `Mmm dd yyyy`. Here, `Mmm` is the month in characters, (Jan, Feb, etc.); `dd` is the day in the form of a pair of characters 1 to 31, where single-digit days are preceded by a blank; and yyyy is the year as four digits (such as 2021).
`__TIME__`	The time at which the source file was compiled, as a character string literal in the form `hh:mm:ss`, which is a string containing the pairs of digits for hours, minutes, and seconds separated by colons.
`__LINE__`	The line number of the current source line as a decimal integer literal. (Note: This macro is superseded by the functionality of C++'s `<source_location>`, as explained shortly.)
`__FILE__`	The name of the source file as a character string literal (superseded by `<source_location>`).
`__func__`[5]	The name of the current function as a character string literal (superseded by `<source_location>`).
`__cplusplus`	A number of type `long` that corresponds to the highest version of the C++ standard that your compiler supports. This number is of the form yyyy`mm`, where yyyy and `mm` represent the year and month in which that version of the standard was approved. At the time of writing, possible values are `199711` for nonmodern C++, `201103` for C++11, `201402` for C++14, `201703` for C++17, and `202002` for C++20. Compilers may use intermediate numbers to signal support for earlier drafts of the standard as well. More fine-grained testing for supported language features may be accomplished by the feature test macros we discussed in the previous section.

Note that each of the macro names in Table A-3 start, and most end, with two underscore characters. You can use the date and time macros to record when your program was last compiled with a statement such as this:

```
std::cout << "Program last compiled at " << __TIME__ << " on " << __DATE__ << std::endl;
```

When this statement is compiled, the values displayed by the statement are fixed until you compile it again. Thus, the program outputs the time and date of its last compilation. These macros can be useful for use in either about screens or log files.

The `__LINE__` and `__FILE__` macros expand to reference information relating to the source file. You can modify the current line number using the `#line` directive, and subsequent line numbers will increment from that. For example, to start line numbering at 1000, you would add this directive:

```
#line 1000
```

[5] Technically, `__func__` is not a macro, but a `static` local variable that is implicitly defined within every C++ function body. We decided to list it in Table A-3 anyway because you often use it together with the macros `__LINE__` and `__FILE__`.

You can use the `#line` directive to change the string returned by the `__FILE__` macro. It usually produces the fully qualified filename, but you can change it to whatever you like. Here's an example:

```
#line 1000 "The program file"
```

This directive changes the line number of the next line to 1000 and alters the string returned by the `__FILE__` macro to `"The program file"`. This doesn't alter the filename, just the string returned by the macro. Of course, if you just wanted to alter the apparent filename and leave the line numbers unaltered, the best you can do is to use the `__LINE__` macro in the `#line` directive:

```
#line __LINE__ "The program file"
```

It depends on the implementation what exactly happens after this directive. There are two possible outcomes: either the line numbers remain unaltered or they are all decremented by one (it depends on whether the value returned by `__LINE__` takes the line on which the `#line` directive appears into account).

You may wonder why you would want to use the `#line` directive to change the line number and/or filename. The need for this is rare, but one example is a program that maps some other language into C or C++. An original language statement may generate several C++ statements, and by using the `#line` directive, you can ensure that C++ compiler error messages identify the line number in the original code, rather than the C++ that results. This makes it easier to identify the statement in the original code that is the source of the error.

■ **Tip** The `<source_location>` module of C++20 offers a better, more powerful alternative to the `__LINE__` and `__FILE__` macros. The following snippet shows a typical use:

```
void logError(std::string_view errorMessage,
       std::source_location location = std::source_location::current())
{
    std::cerr << std::format("{}:{}:{} - An unexpected error occurred in {}: {}",
       location.file_name(), location.line(), location.column(),
       location.function_name(), message) << std::endl;
}
```

When invoked from the `main()` function in `ExA_03.cpp` (available online), `logError("OOPS!");` produces output of the following form: "`ExA_03.cpp:16:4 - An unexpected error occurred in main(): OOPS!`". Because default argument values are evaluated at the calling site, line 16 refers to the line at which `logError()` is invoked from within `main()`—and so *not* to the line at which the `logError()` function itself is defined. With `__LINE__`, `__FILE__`, and `__func__`, you'd have to turn `logError()` into a macro as well to obtain a similar result. Only then you wouldn't have access to the column number of the source location (at least not in a standardized way). Another advantage of the new `<source_location>` functionality is that you can readily store `std::source_location` objects and pass them around. One interesting use, for instance, could be to store `std::source_location` objects in your exception objects (see also Exercise A-3 at the end of this appendix).

Using the assert() Macro

The `assert()` preprocessor macro is defined in the Standard Library header `<cassert>`. It enables you to test logical expressions in your program. Including a line of the form `assert(expression)` by default results in code that causes the program to be terminated with a diagnostic message if `expression` evaluates to `false`. We can demonstrate this with this simple example:

```
// ExA_04.cpp
// Demonstrating assertions
import <iostream>;
import <format>;
#include <cassert>

int main()
{
  int y {5};

  for (int x {}; x < 20; ++x)
  {
    std::cout << std::format("x = {}\ty = {}", x, y) << std::endl;
    assert(x < y);
  }
}
```

You should see an assertion message in the output when the value of `x` reaches `5`. When an assertion is triggered—which in our example occurs when `x < y` evaluates to `false`—the program is terminated. Concretely, the `assert()` macro calls the `std::abort()` function from the Standard Library, which effectively terminates the program immediately. Right before terminating, `assert()` outputs a diagnostic message on the standard error stream, `cerr`. The message typically contains the condition that failed and also the filename and line number in which the failure occurred. This is particularly useful with multifile programs, where the source of the error is pinpointed exactly.

Assertions are often used for critical conditions in a program where, if certain conditions are not met, disaster will surely ensue. You would want to be sure that the program wouldn't continue if such errors arise. You can use any logical expression as the argument to the `assert()` macro, so you have a lot of flexibility.

Using `assert()` is simple and effective, and when things do go wrong, it provides sufficient information to pin down where the program has terminated.

■ **Tip** Some debuggers, in particular those integrated into graphical IDEs, allow you to pause each time an assertion is triggered, right before the application terminates. This greatly increases the value of assertions during debugging sessions.

■ **Caution** `assert()` is for detecting programming errors, not for handling errors at runtime. It is for asserting facts that you, as a programmer, are 110% convinced should never, ever occur. The logical expression of an assertion should thus never be based on something beyond your control (such as whether opening a file succeeds). Your program should include code to handle any and all error conditions that might be expected to occur occasionally—however unlikely their occurrence may be. As seen in Chapter 16, exceptions are an effective means of signaling and/or handling such runtime errors in C++.

Switching Off assert() Macros

You can switch off the preprocessor assertion mechanism when you recompile the program by defining NDEBUG at the beginning of the program file:

```
#define NDEBUG
```

This causes all assertions in the translation unit to be ignored. If you add this #define at the beginning of ExA_04.cpp, you'll get output for all values of x from 0 to 19 and no diagnostic message. Note that this directive is effective only if it's placed *before* the #include directive for <cassert>.

■ **Tip** Most compilers also allow you to define macros such as NDEBUG globally for all source and header files at once (for instance, by passing a command-line argument or by filling in some field in your IDE's configuration windows). Often NDEBUG is defined that way for fully optimizing so-called "release" configurations, but not for the configurations that are used during debugging. Consult your compiler's documentation for more details.

■ **Caution** If NDEBUG is defined, the logical expression passed to assert() is not evaluated. Because it is then removed by the preprocessor, this expression is not even part of the compiled code anymore. Because assertions are typically turned off for optimizing compilations, evaluation of the logical expression should thus never cause any side effects.

Static Assertions

Static assertions, unlike the assert() macro, are part of the C++ language. That is, they are no Standard Library addition but built into the language. The assert() macro is for checking conditions *dynamically, at runtime*, whereas static assertions are for checking conditions *statically, at compile time*. You already encountered an example use of static assertions in Chapter 21, where we added assertions to verify that a given type models a given concept—a technique that is useful both when developing a new class type and when developing a new concept.

In general, a static assertion is a statement of either of the following forms:

```
static_assert(constant_expression);
static_assert(constant_expression, error_message);
```

static_assert is a keyword, constant_expression must produce a result at compile time that can be converted to type bool, and error_message is an optional string literal. If constant_expression evaluates to false, then the compilation of your program should fail. The compiler will abort the compilation and output a diagnostics message that contains error_message if you provided it. If you did not specify an error_message, the compiler will generate one for you (usually based on constant_expression). When constant_expression is true, a static assertion does nothing.

The compiler needs to be able to evaluate constant_expression during compilation. This limits the range of expressions you can use. Typical static_assert() expressions consist of literals, const variables that are initialized by literals, macros, the sizeof() operator, template arguments, and so on. A static assertion cannot, for instance, check the length() of an arbitrary std::string or use the value of a function argument or any other non-const variable—such expressions can be evaluated only at runtime.

■ **Note** With every new edition of the C++ standard, the range of expressions and functions that compilers should be able to evaluate at compile time is increasing. You can define all kinds of functions, variables, and even lambda expressions that it should be able to evaluate statically simply by adding the `constexpr` keyword to their declaration. Naturally, such functions remain bound by certain restrictions; not everything is possible at compile time. The use of `constexpr`, however, is beyond the scope of this book.

As a brief example, suppose that your program does not support 32-bit compilation, for instance because it needs to address more than 2GB of memory to process larger data sets. Then you could put the following static assertion anywhere in your source file:

```
static_assert(sizeof(int*) > 4, "32-bit compilation is not supported.");
```

Adding this static assertion will thus ensure that you cannot inadvertently compile as a 32-bit program.

■ **Note** Prior to C++20, static assertions were often used to restrict type parameters of templates, typically in an attempt to produce more legible error messages in case these templates were ever inadvertently instantiated for unsupported type arguments. (See Chapter 21 for a discussion on how notoriously unclear and verbose template instantiation errors can be.) Here is an example:

```
template<typename T>
T average(const std::vector<T>& values)
{
  static_assert(std::is_arithmetic_v<T>,
              "Type parameter for average() must be arithmetic.");
  // ... Code that applies operators + and / to T values
}
```

`std::is_arithmetic_v<T>` is a constant expression based on a type trait from the `<type_traits>` module (you encountered type traits in Chapter 21). It evaluates to `false`, thus triggering the static assertion, whenever the template is instantiated for a template type argument `T` that is anything but a floating-point or integral type.

From Chapter 21, however, you of course know that type constraints and requires clauses are far more powerful and effective for constraining template type parameters. Here is how you would therefore write the previous example in C++20:

```
// There is no std::arithmetic<> concept (yet?), so we define one ourselves

template <typename T>
concept Arithmetic = std::integral<T> || std::floating_point<T>;

template <Arithmetic T>
T average(const std::vector<T>& values)
{
  // ... Code that applies operators + and / to T values
}
```

Including Files

The final preprocessing directive we introduce is `#include`. The `#include` directive takes the contents of a given file and injects that into the current file. To illustrate this, you can create a file called `inclusivity.quote` with the following three lines of code:

```
std::cout << "We are trying to construct a more inclusive society.\n";
std::cout << "We are going to make a country in which no one is left out.\n"
          << "\t\t\t\t\t- Franklin D. Roosevelt" << std::endl;
```

You can then inject these three lines into an actual program with an `#include` directive as follows:

```
// ExA_05.cpp
// Introducing the #include directive
import <iostream>;

int main()
{
#include "inclusivity.quote"
}
```

In the translation unit that the preprocessor hands over to the compiler for this source file, the `#include` directive will be gone. It's replaced with the content of the `inclusivity.quote` file. When executed, `ExA_05` therefore outputs the following inspirational and all-too-relevant quote on inclusivity:

```
We are trying to construct a more inclusive society.
We are going to make a country in which no one is left out.
                              - Franklin D. Roosevelt
```

Of course, the way we used `#include` in `ExA_05` is not how you would normally use this directive. You normally use `#include` to include header files. You also normally put all your `#include` directives near the top of a file, and not in the middle of a function body. But `ExA_05` does nicely illustrate what an `#include` directive is: a crude, low-level construct that instructs the preprocessor to look for a file with a given name, take its contents, and copy-paste it into the translation unit at the exact spot where it found the `#include` directive. Nothing more, nothing less. As we will see later, this has a significant impact on the remainder of the translation process, as well as on how you should compose and structure your header files. But first, some additional notes on the syntax of the `#include` directive.

Double Quotes vs. Angle Brackets

Here is how you included your own source file named `inclusivity.quote` into the translation unit for `ExA_05.cpp`.

```
#include "inclusivity.quote"
```

This syntax differs slightly from the syntax you used to include Standard Library headers before, which looks as follows (seen first in Chapter 2):

```
#include <cmath>
```

The concrete difference between specifying a filename between double quotes or between angled brackets lies in the locations in which the preprocessor will search for a file with that name. The precise locations the preprocessor uses for this are implementation-dependent and should be described in your compiler documentation. Usually, though, the idea is that you use angle brackets for headers of external libraries, and double quotes for files that are stored next to the current source file.

```
#include <some_external_header.h>
#include "some_local_header.h"
```

Usually, when a filename is between double quotes, the preprocessor first searches the current directory (typically either the directory that contains the source file that is being compiled, or one of its parent directories). Eventually, though, if the file is not found there, the preprocessor is required to fall back to searching the same directories that it would search had the name been between angled brackets.

When a name is between angle brackets, then, the preprocessor by default typically only searches the directories that it knows contain the Standard Library headers, although sometimes this is automatically combined with directories that contain headers native to the target operating system as well. Most compilers allow you to provide other directories for the preprocessor to search as well, though. The idea is that these so-called additional *include directories* then contain the headers of any external (often third-party) library that your program may rely on.

■ **Tip** Most third-party libraries make their APIs available through header files that are located in a directory called `include`.

If a header file is located in some subdirectory, either relative to the current working directory or to any of the additional `include` directories, you normally put the relative path for the header file between quotes or braces. Here is an example.

```
#include "subdirectory/in/current_directory/myheader.h"
#include <subdirectory/in/external_library_include_directory/some_header.h>
```

Understanding Header Inclusion

Later in this appendix we will show you how to correctly compose header files of your own. Before we can do this, though, you first need to understand the limitations (in particular, the so-called *one definition rule*) and some more underlying concepts (such as *external linkage*). Nevertheless, we believe it is beneficial to already give you an intuitive feeling of how header inclusion works here. This should make it easier for you to follow the upcoming, more theoretical, sections that we need to explain these limitations and concepts.

Consider this familiar `#include` directive, which you have used many times already to gain access to the Standard Library's basic mathematical functions (see Chapter 2 for an overview).

```
#include <cmath>
```

In a first step, the preprocessor replaces this `#include` directive with the contents of the `<cmath>` Standard Library header. Where exactly this header file is located is implementation defined.

Like with macro text replacement before, though, any included body of text is immediately preprocessed as well, which often leads to the contents of more files being injected into the translation unit. And so on, until no more preprocessing directives are left.

This is also what happens after including `<cmath>`. Even though `<cmath>` is arguably one of the more straightforward Standard Library headers[6], a `<cmath>` header typically contains `#include` directives for several more headers, which in turn rely on even more headers, and so on. In the Standard Library implementation we looked at, a single `#include` directive for `<cmath>` alone, fully expanded, resulted in about 6,000 (!) lines of code (over 10,000 if you include code comments) being injected into our translation unit. Which is massive, given that you typically only want to call one or two simple math functions, such as `std::abs()` or `std::sqrt()`.

This immediately illustrates one of the primary disadvantages of `#include` directives: they generate a tremendous amount of work for the preprocessor, compiler, and linker, all of which have to process code over and over again for countless C++ entities, most of which are typically not even used by your application. The good news, though, is that, even though this obviously puts pressure on your compilation times, the impact of `#includes` on the eventual size of your executable is usually limited. For the most part, code for entities that you do not use does not make it into the final binary.

This should give you enough of a feeling of how header inclusion works. In short, including even a single header can often inject thousands of lines of code into a translation unit, consisting of hundreds of entity declarations and definitions. The exact implications on which declarations and definitions you are therefore allowed to put in your own header files, and how you should do this robustly, is something we can only properly explain once you know more about the one definition rule and the different types of linkage.

The One Definition Rule

The *one definition rule* (ODR) is an important concept in C++. Despite its name, ODR is not really one single rule; it's more like a set of rules. Our exposition of these rules won't be exhaustive, nor will it use the formal jargon you'd need for it to be 100% accurate. Our main intent is to familiarize you with the general ideas behind the ODR restrictions. This will help you to better understand how to organize the code of your program in header and source files and to decipher and resolve the compiler and linker errors you'll encounter when you violate an ODR rule.

ODR Within a Translation Unit

In a given translation unit, no variable, function, class type, enumeration type, or template must ever be *defined* more than once. You can have more than one *declaration* for any such entity, but there must never be more than one *definition* that determines what it is and causes it to be created. If there's more than one definition within the same translation unit, the code will not compile.

■ **Note** A *declaration* introduces a name into a scope. A *definition* not only introduces the name but also defines what it is. In other words, all definitions are declarations, but not all declarations are definitions.

[6] Although you should never underestimate the complexity and ingenuity that goes into defining an efficient, numerically stable mathematical primitive!

To familiarize yourself with the error messages that ODR violations produce, you can try to compile the following program.

```
// ExA_06.cpp
// Defining the same function twice
import <iostream>;
import <format>;

double power(double x, int n);
double power(double x, int n);  // Redundant declaration (harmless)

int main()
{
  for (int i {-3}; i <= 3; ++i)    // Calculate powers of 8 from -3 to +3
    std::cout << std::format("{:10}\n", power(8.0, i));
  std::cout << std::endl;
}

double power(double x, int n)   // A first definition
{
  if (n == 0)      return 1.0;
  else if (n > 0)  return x * power(x, n - 1);
  else /* n < 0 */ return 1.0 / power(x, -n);
}

double power(double x, int n);  // Another redundant declaration (harmless)

double power(double x, int n)   // A second, more efficient definition (error!)
{
  if (n < 0) return 1.0 / power(x, -n); // Deal with negative exponents
  if (n == 0) return 1.0;               // Base case of the recursion
  const double y{ power(x, n / 2) };    // See Exercise 8-8 for an explanation
  return y * y * (n % 2 == 1 ? x : 1.0);
}
```

`ExA_06.cpp` contains a number of declarations and definitions for one and the same function: `power()`. You may recognize this function from Chapter 8. We refer to Exercise 8-8 in particular for an explanation of the less-obvious recursive definition at the bottom of `ExA_06.cpp`.

From Chapter 8, you'll also recall why `main()` can call `power()`, even though the definition of this function is not known to the compiler when it processes the body of `main()` (a compiler processes source top-to-bottom). To invoke a function, the compiler only needs to know its prototype, which is contained in non-defining declarations as well. Remember this, as this will prove important in understanding how the separation of code in header and source files works (this, in combination with *external functions*, which we discuss in one of the next sections).

While it's certainly odd to declare the same function more than once like we did in `ExA_06.cpp`, it is perfectly allowed by C++. You can have as many *declarations* for the same function as you want within a translation unit, as long as they are all equivalent, and no more than one of these declarations is a *definition*.

Obviously, the problem with `ExA_06.cpp` thus lies with its two definitions of `power(double, int)`. How is the compiler to know which of these definitions it should generate machine code for in the object file? The naïve implementation at the top? Or the more efficient one at the bottom? And so, compilation of `ExA_06.cpp` fails with an error—and rightfully so!

Of course, only a true scatterbrain would explicitly define the same entity twice within the same file. But duplicate definitions do make it into the same translation unit at times by (indirectly) including an improperly created header file more than once. We discuss the techniques you should use to prevent this later in this appendix.

ODR Within a Program

The ODR rules do not only apply within one translation unit. They also stipulate that regular functions and variables must be defined once and only once *within an entire program*. As a rule, no two definitions of the same function or variable are allowed, even if they're identical and appear defined in different translation units. Naturally, violations against this aspect of the ODR rules are only discovered during the linking stage of translation.

Exceptions to this rule are *inline functions* and *inline variables*. In fact, for inline functions and inline variables that are not defined in a module file, a definition *must* appear once and only once *in every translation unit* that uses them. All definitions of a given inline function or variable within different translation units of a program have to be identical, though. One traditionally uses inline functions and variables to define variables or functions in header files, something which with regular functions and variables would almost certainly lead to ODR violations. We show you how to define inline entities later, when we are ready to discuss header files.

The program-wide ODR rules for class types (see Chapter 12), enumeration types (Chapter 3), and templates (Chapters 10 and 17) are similar to those of inline functions and variables. That is, unless they are defined in a module file, one and only one definition is required for these entities *within every translation unit* that uses them. Note that, in a way, the compiler does preserve an ODR-like behavior for templates by instantiating each template only once for any given combination of template arguments.

While multiple definitions of the same class type are thus allowed, its member functions and variables do have to obey the same ODR rules as regular functions and variables. In other words, non-inline class members must have only one single definition within the entire program. This is one of the reasons why in non-modular code you generally put the definitions for a class's members in a source file, while the class definition itself is placed in the corresponding header file. We will see an example of this later.

ODR and Modules

In a fully modular program, you are far less likely to ever run into issues with ODR. In fact, the only way you can get ODR violations in modular code is when you export the same entity from multiple modules—something you should be able to avoid by properly organizing your names in namespaces (see Chapter 11).

A first, obvious, reason that ODR violations are far less likely with modules is that `import` declarations are fundamentally different from `#include` directives. Any definition you place in a header gets copy-pasted into every translation unit that includes it, either directly or indirectly. Because this inherently copies declarations many times over, you clearly always have to be watchful for ODR violations in the non-modular setting. The content of a module interface file, however, is never duplicated. An `import` declaration is part of the C++ language, and therefore does not have to be processed and removed by the preprocessor. It simply tells the C++ compiler to make the declarations from a given module visible and/or reachable within the consuming translation unit (see Chapter 11); no definitions are ever copied when importing a module. Any duplicate definition in modular code will therefore be one by your hand.

The second reason that you are far less likely to run into ODR issues with modular code stems from the fact that non-exported entities in a named module `A` are always different from non-exported entities in module `B`, even if these entities appear to have the exact same name or signature. Formally, module-local entities are said to have *module linkage*, whereas local entities in non-modular source files generally have *external linkage* by default. We discuss these and other types of linkage further in the next section.

The Different Types of Linkage

Entities defined in one translation unit often need to be accessed from code in another translation unit, even in non-modular code. Functions and class types are obvious examples of where this is the case, but you can have others—variables defined at global scope that are shared across several translation units, for instance, or the definitions of type aliases or enumeration types. Because the compiler processes one translation unit at a time, such references can't be resolved by the compiler. Only the linker can do this when all the object files from the translation units in the program are available.

The way that a name in a translation unit is handled in the compile/link process is determined by a property called *linkage*. The linkage of a name expresses where in the program code the entity it represents can be defined and/or accessed. Every name that you use in a program either has linkage or doesn't. A name has linkage when you can use it to access something in your program that is *outside* the scope in which the name is declared. If this isn't the case, it has no linkage. If a name does have linkage, then it can have *internal linkage*, *module linkage*, or *external linkage*.

Determining Linkage for a Name

The linkage for each name in a translation unit is determined *after* the contents of any header files have been inserted into the source file that is the basis for the translation unit. The linkage possibilities have the following meanings:

- *Internal linkage*: The entity that the name represents can only be accessed from within the same translation unit. For example, the names of non-inline, non-exported, `const`-qualified variables defined at global or namespace scope have internal linkage by default.
- *External linkage*: A name with external linkage can be accessed from other translation units in addition to the one in which it is defined. In other words, the entity that the name represents can be shared and accessed throughout the entire program. In non-modular code, most names that are declared at global or namespace scope have external linkage by default—including those of functions, enumerations, classes, templates, and inline or non-`const` variables. Naturally, all entities that are exported from a module have external linkage as well.
- *Module linkage*: An entity name with module linkage is accessible only in translation units of the same named module. The name of any non-exported entity that is declared at global or namespace scope within the purview of a module file (in other words, anywhere after the module file's optional global module fragment; see Chapter 11) has module linkage by default.
- *No linkage*: When a name has no linkage, the entity it refers to can be accessed only from within the scope that applies to the name. All names that are defined within a block—local names, in other words—have no linkage.

External Functions

In a program made up of several files, the linker establishes (or *resolves*) the connection between a function call in one source file and the function definition in another. When the compiler compiles a *call* to the function, it only needs the information contained in a function prototype to create the call. The compiler either extracts this prototype from a (precompiled) module interface file (see Chapter 11), or from any function *declaration* present in the current translation unit itself (often after including it from a header file).

Either way, the compiler doesn't care whether the function's *definition* occurs in the same translation unit, or in any other translation unit. If a function is not defined within the translation unit in which it is called, the compiler flags the call as external and leaves it for the linker to sort out.

We should clarify this with an example. For this, we'll adapt `ExA_01.cpp` and move one of the definitions of its `power()` function to a different translation unit. The other definition of `power()` we simply discard (although it may be instructive for you to move that definition to a second source file as well, and study the ODR-related linker errors you then get during linking).

```
// ExA_07.cpp
// Calling external functions
import <iostream>;
import <format>;

double power(double x, int n);   // Declaration of an external power() function

int main()
{
  for (int i {-3}; i <= 3; ++i)  // Calculate powers of 8 from -3 to +3
    std::cout << std::format("{:10}\n", power(8.0, i));
  std::cout << std::endl;
}
```

Even though `power()` is called by `main()`, no definition of this function is present in the `ExA_07` translation unit. But that's okay. Like we said, all the compiler needs to carry out a call to `power()` is its prototype. Because function names have external linkage by default, the compiler simply makes note of a call to an externally defined `power()` function inside the object file for the `ExA_07` translation unit. It then becomes the linker's job to hook up—or *link*—this call with a definition. If the linker doesn't find the appropriate definition in one of the other translation units of the program, it will signal this as a translation failure.

To make the `ExA_07` program translate correctly, you'll thus need a second translation unit with the definition of `power()`. We of course pick the more efficient implementation and discard the slower one.

```
// Power.cpp
// The power function called from ExA_07.cpp is defined in a different translation unit
double power(double x, int n)
{
  if (n < 0) return 1.0 / power(x, -n); // Deal with negative exponents
  if (n == 0) return 1.0;               // Recursion base case
  const double y{ power(x, n / 2) };    // See Exercise 8-8 for an explanation
  return y * y * (n % 2 == 1 ? x : 1.0);
}
```

By linking the object files of the `ExA_07` and `Power` translation units, you obtain a program that is otherwise completely equivalent to that of `ExA_01` (after first removing the slower definition of `power()` there as well, of course).

Notice that in order to use the `power()` function, we still had to supply the compiler with a prototype in the beginning of `ExA_07.cpp`. It would be very impractical if you had to do this explicitly for every externally defined function, and that you moreover had to repeat these same prototypes over and over again in every source file that wants to call them. This is why, in non-modular code, function prototypes are typically gathered in header files, which you can then conveniently `#include` into your translation units. Here is a sneak preview of how this works.

```
// Power.h
// Your first header file containing a declaration of an external power() function
double power(double x, int n);

// ExA_07A.cpp
// Calling external functions that are declared in a header file
import <iostream>;
import <format>;

#include "Power.h"

int main()
{
  for (int i {-3}; i <= 3; ++i)  // Calculate powers of 8 from -3 to +3
    std::cout << std::format("{:10}\n", power(8.0, i));
  std::cout << std::endl;
}
```

The only difference between ExA_07A and ExA_07 is that we moved the declaration of power() to a header file, Power.h, which we subsequently included in ExA_07A.cpp. By including Power.h, the preprocessor obviously produces the exact same translation unit for ExA_07A.cpp as it did for ExA_07.cpp, and so the compiler does not even notice the difference between both programs. You, the developer, however, now only have to type the prototype of power() twice—once for the declaration in the header file, and once for the definition in the source file—which is clearly a step forward, compared to having to type it once in every source file in which you want to call the function. Of course this gain is even greater if you need copies of entire class definitions in every translation unit that uses the class, but we'll get to that.

This naïve header file, Power.h, is perfectly correct and safe (albeit somewhat unconventional), yet only because it does not contain any definitions. If your header does contain definitions (class definitions, template definitions, inline definitions of variables or functions, and so on), things become slightly more involved. Without additional precautions, header inclusion would then quickly lead to ODR violations, whereby the same definition is inadvertently included in the same unit multiple times. Later in this appendix we explain how you can avoid such issues.

Before we show you how to create more advanced header files, though, we first continue with the various types of linkage, moving on now from external functions to external variables.

External Variables

Suppose that in ExA_07.cpp, you wanted to replace the magic constants -3 and 3 using an externally defined variable power_range, like so:

```
  for (int i {-power_range}; i <= power_range; ++i)     // Calculate powers of 8
    std::cout << std::format("{:10}\n", power(8.0, i));
```

The first step is to create an extra source file called Range.cpp containing the variable's definition:

```
// Range.cpp
int power_range{ 3 };            // A global variable with external linkage
```

Non-const variables have external linkage by default, just like functions do. So other translation units will have no problem accessing this variable. The interesting question, though, is this: how do you declare a variable in the consuming translation unit without it becoming a second definition? A reasonable first attempt would be this:

```
// ExA_08.cpp
// Using an externally defined variable
import <iostream>;
import <format>;

double power(double x, int n);  // Declaration of an external power() function
int power_range;                // Not an unreasonable first attempt, right?

int main()
{
  for (int i {-power_range}; i <= power_range; ++i)  // Calculate powers of 8
    std::cout << std::format("{:10}\n", power(8.0, i));
  std::cout << std::endl;
}
```

The compiler will have no problem with this declaration of power_range. The linker, however, will signal an error! We recommend you give this a try as well to familiarize yourself with this error message. In principle (linker error messages do not always excel in clarity), you should then be able to deduce that we have supplied two distinct definitions for power_range: one in Range.cpp and one in ExA_08.cpp. This, of course, violates the one definition rule!

The underlying problem is that our declaration of the power_range variable in ExA_08.cpp is not just any old variable declaration; it's a variable *definition*:

```
int power_range;
```

In fact, you might've already known this would happen. Surely, you'll remember that variables generally contain garbage if you neglect to initialize them. Near the end of Chapter 3, however, we also covered *global* variables. And global variables, as we told you, will be initialized with zero, even if you omit the braced initializer from their definition. In other words, our declaration of the global power_range variable in ExA_08.cpp is equivalent to the following definition:

```
int power_range {};
```

And ODR does not allow for two definitions of the same variable. The compiler therefore needs to be told that the definition for the global variable power_range will be external to the current translation unit, ExA_08. If you want to access a variable that is defined *outside* the current translation unit, then you must declare the variable name using the extern keyword:

```
extern int power_range;   // Declaration of an externally defined variable
```

This statement is a declaration that power_range is a name that is defined elsewhere. The type must correspond exactly to the type that appears in the definition. You can't specify an initial value in an extern declaration because it's a declaration of the name, not a definition of a variable. Declaring a variable as extern implies that it is defined in another translation unit. This causes the compiler to mark the use of the externally defined variable. It is the linker that makes the connection between the name and the variable to which it refers.

Within any block in which this declaration appears, the name power_range refers to the variable defined in another file. The declaration can appear in any translation unit that needs access to power_range. You can place the declaration either at global scope so that it's available throughout the entire translation unit, or within a block, in which case it is available only within that local scope.

■ **Note** You're allowed to add extern specifiers in front of function declarations as well. For example, in ExA_08.cpp you could've declared the power() function with an explicit extern specifier to call attention to the fact that the function's definition will be part of a different translation unit:

```
extern double power(double x, int n);
extern int power_range;
```

While entirely optional, our advice is to add extern in front of any declaration for an externally defined function, unless that declaration occurs in a header or module interface file. Doing so is not just consistent, but also far clearer for the person reading the code.

---

const Variables with External Linkage

Given its nature, you'd of course want to define the power_range variable from Range.cpp of ExA_08 as a global constant, rather than as a modifiable global variable:

```
// Range.cpp
const int power_range {3};
```

A const variable, however, has internal linkage by default, which makes it unavailable in other translation units. You can override this by using the extern keyword in the definition:

```
// Range.cpp
extern const int power_range {3}; // Definition of a global constant with external linkage
```

The extern keyword tells the compiler that the name should have external linkage, even though it is const. When you want to access power_range in another source file, you must declare it as const and external:

```
extern const int power_range;     // Declaration of an external global constant
```

You can find this in a fully functioning example in ExA_08A.

Global variables can be useful for constant values that you want to share because they are accessible in any translation unit. By sharing constant values across all of the program files that need access to them, you can ensure that the same values are being used for the constants throughout your program. As with external functions, though, if a global variable is required in multiple non-modular files, you generally place them in a header file. You'll see an example of this later in this appendix.

■ **Tip** As said also in Chapter 3, coding and design guidelines typically dictate that global variables are to be avoided. Global constants, though—global variables declared with the const keyword —are a noble exception to this rule. It is recommended to define all your constants only once (in a header or module file), and global variables are perfectly suited for that.

Internal Names

If there's a way to specify that names should have external linkage, surely there must be ways to specify that they should have internal linkage as well, right? And of course there are. Two even. Yet neither is the one you'd expect!

Let's first illustrate when and why you'd need this possibility. Perhaps you noticed that upon every recursive call of power() in ExA_07, the function checks whether its argument n is positive or negative. This is somewhat wasteful because the sign of n, of course, never changes. One option is to rewrite power() in the following manner:

```
// Power.cpp
// A slightly optimized power function

double localHelper(double x, unsigned n)
{
  if (n == 0) return 1.0;                  // Recursion base case
  const double y{ localHelper(x, n / 2) }; // See Exercise 8-8 for an explanation
  return y * y * (n % 2 == 1 ? x : 1.0);
}

double power(double x, int n)
{
  return n >= 0 ? localHelper(x, static_cast<unsigned>(n))
              : 1.0 / localHelper(x, static_cast<unsigned>(-n));
}
```

The power() function itself is now no longer recursive. Instead, it calls the recursive helper function localHelper(), which is defined to work only for positive (unsigned) exponents n. Using this helper, it's easy to rewrite power() in such a way that it checks whether the given n is positive only once.

In this case, localHelper() could in principle be a useful function in its own right—best renamed then to become an overload of power() specific for unsigned exponents. For argument's sake, however, suppose we want localHelper() to be nothing more than a local helper function, one that is only to be called by power(). You'll find that the need for this occurs quite often; you need a function to make your local code clearer or to reuse within one specific translation unit, but that function is too specific for it to be exported to the rest of the program for reuse.

Our localHelper() function currently has external linkage as well, just like power(), and can therefore be called from within any translation unit. Worse, the one definition rule implies that no other translation unit may define a localHelper() function with the same signature anymore either! If all local helper functions always had external linkage, they'd all need unique names as well, which would make it very hard indeed to avoid name conflicts in larger programs.

What we need is a way to tell the compiler that a function such as localHelper() should have internal linkage rather than external linkage. Given that extern gives names external linkage, an obvious attempt would be to add an intern specifier in front of the definition. And that just might've worked, if not for the little detail that there's no such keyword in C++. Instead, in the (very) old days, the way to mark a name (function or variable name) for internal linkage was by adding the static keyword. Here's an example:

```
static double localHelper(double x, unsigned n) // localHelper() now has internal linkage
{
  if (n == 0) return 1.0;                  // Recursion base case
  const double y{ localHelper(x, n / 2) }; // See Exercise 8-8 for an explanation
  return y * y * (n % 2 == 1 ? x : 1.0);
}
```

While this notation will still work, this use of the `static` keyword is no longer recommended. The only reason that this syntax, which originates from C, is not (or no longer, for those who know their history) deprecated or removed from the C++ Standard is that you'll still find it a lot in legacy code. Nevertheless, the recommended way to define names with internal linkage in C++ is through unnamed namespaces.

■ **Caution** Never use `static` anymore to mark names that should have internal linkage; always use unnamed namespaces instead.

Unnamed Namespaces

You declare an *unnamed namespace* with the following code:

```
namespace
{
   // Code in the namespace, functions, etc.
}
```

Declaring an entity within an unnamed namespace effectively has the same effect as declaring it as `static` in the global namespace, except that `static` can only be applied to variables or functions. Within a translation unit, you refer to entities of its unnamed namespace as if they were defined in the global namespace (even if they technically are not). Here is how you would therefore normally create a local helper function in the `Power.cpp` source file of our running example (the result is available from `ExA_09`).

```
// Power.cpp
// A local helper function with internal linkage
namespace
{
  double localHelper(double x, unsigned n)   // localHelper() has internal linkage
  {
    if (n == 0) return 1.0;                  // Recursion base case
    const double y{ localHelper(x, n / 2) }; // See Exercise 8-8 for an explanation
    return y * y * (n % 2 == 1 ? x : 1.0);
  }
}

double power(double x, int n)               // power() has external linkage
{
  return n >= 0 ? localHelper(x, static_cast<unsigned>(n))
                : 1.0 / localHelper(x, static_cast<unsigned>(-n));
}
```

Entities with internal linkage are never accessible from outside the translation unit. Similar to the fact that the parameter names `x` in `localHelper()` and `power()` refer to two distinct entities, any function declaration with prototype `double localHelper(double, unsigned)` in a translation unit other than `Power` will now no longer name the same entity. You can try this by altering the main source file of `ExA_07` as follows:

```
// ExA_09A.cpp
// Attempting to call a function with internal linkage from a different translation unit
import <iostream>;
import <format>;

double localHelper(double x, unsigned n); // Declares an external localHelper() function

int main()
{
  for (unsigned i {0}; i <= 5; ++i)  // Calculate positive powers of 8
    std::cout << std::format("{:10}\n", localHelper(8.0, i));
  std::cout << std::endl;
}
```

The declaration in `ExA_09A` now declares an external function `localHelper()`—a function for which the linker will not find a definition.

■ **Tip** All names declared inside an unnamed namespace have internal linkage (even names defined with an `extern` specifier…). If a function, global variable, class, enumeration, or template is not supposed to be accessible from outside a particular translation unit, you should therefore always define it in an unnamed namespace, especially in non-modular code (see also the next section). Using a `static` specifier for this purpose is no longer recommended (and was only ever possible for function and variable definitions anyway).

Module-Local Names

Each module in a way forms its own self-contained little universe in which the ODR applies. Naturally, names of exported entities have external linkage, which means that two modules exporting the same entity constitutes an ODR violation (technically even if you never import both modules in the same file). But names of non-exported entities by default have module linkage if they are declared at global or namespace scope in the purview of a module file; not external linkage. This means that if multiple modules for example each define a module-local function with the same prototype, these are not seen as the same entity by the linker, and thus will not lead to a violation of any ODR rule. The following example illustrates the implications of module linkage.

```
// ModuleA.cppm
export module A;
void logError(std::string_view error) { /* ... */ }
// Code that uses the module-local logError() in case of error...
// Other module files of module A can call this same function.
// No additional declaration is required in module implementation files
// because this interface file is implicitly imported there (see Chapter 11).
// In module partition files either an 'import A;' declaration
// or an additional declaration of logError() is required, though.

// ModuleB.cppm
export module B;
void logError(std::string_view error) { /* ... */ }
// Code that uses the module-local logError() in case of error...
// Analogous to module A. Because both logError() names have module linkage,
// these functions are seen as independent, different entities,
// and can therefore never cause any ODR issues--not even if you import
// both A and B into the same source file.

// ModuleCSource.cpp
module C;
void logError(std::string_view error) { /* ... */ }
// Code that uses the module-local logError() in case of error...
// Other module files of module C can call this same function,
// provided they add a declaration first.
```

If you only need a particular entity in one module translation unit, you can of course still give its name internal linkage by declaring it in an unnamed namespace. A concrete example for which this would have been applicable is the local `from_roman(char)` helper function in the `from_roman.cpp` source file of `Ex11_02`. You can even imagine that singling internal linkage gives the compiler and linker slightly less work, or even that it may facilitate certain optimizations such as inlining (see also later in this appendix). That notwithstanding, using unnamed namespaces to avoid ODR violations is somewhat less crucial in modular code. After all, you are far less likely to encounter ODR violations by inadvertently defining two identical entities within the same module than you are within an entire program (modules are typically much smaller and under full control of the same developer or team of developers).

Organizing Non-Modular Code

Now that you understand preprocessing directives, the ODR rules, and the different types of linkage, we are finally ready to explain how you correctly organize non-modular code in headers and source files.

■ **Note** If your compiler supports it, we highly recommend you always use modules in new code, and gradually convert existing code to modules whenever possible (the final section of this appendix further reinforces this recommendation). But of course this is not always an option (even as we are writing this, no compiler fully supports modules yet), and even if it is, you will likely still have to work in legacy, non-modular code for a long time to come. So, either way, it for sure remains important to understand how headers and source files interrelate.

We already gave you a sneak peek at how header inclusion works in `ExA_07A`, where we included the following basic header file to avoid having to repeat the prototype of `power()` in every source file in which you want to call this external function.

```
// Power.h
// Your first header file containing a declaration of an external power() function
double power(double x, int n);
```

Including a header file (such as `Power.h`) that does not contain any *definitions* never leads to issues with the one *definition* rule (ODR). Without duplicate definitions, there are no ODR violations—that much is clear. In what follows, we look at the two types of ODR violations that occur if you do add definitions to a header file in a naïve way, and of course we teach you how to avoid these issues.

Preventing Duplication of Header File Contents

For your first non-trivial header file, it is only fitting that we start with the first module file that you ever created: `math.cppm` (see `Ex11_01`). This is how you naïvely convert that module interface file into a header file.

```
// BadMath.h - The start of your very first proper header file
auto square(const auto& x) { return x * x; }    // An abbreviated function template

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

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

If you include this header twice into the same translation unit, though, you of course violate the most obvious ODR rule: no two definitions of the same entity are allowed within the same translation unit, ever. In fact, after preprocessing, you basically end up in exactly the same situation as in `ExA_06` earlier.

```
// ExA_10.cpp
// Including the same definitions twice
import <iostream>;
#include "BadMath.h"
#include "BadMath.h"

int main()
{
  std::cout << square(1.234) << std::endl;
}
```

Of course, only a true scatterbrain (the same scatterbrain that explicitly defined the same function twice in `ExA_06`) would explicitly include the same header twice into the same source file like this. But, as we explained earlier, each header that you include typically includes several more headers, and this process can go on many levels deep. Even in most basic of programs, there is therefore a good chance that you might include header files more than once in the same translation unit indirectly. In some situations this may even be unavoidable.

Adding a definition to a header file therefore clearly opens the door to ODR violations through indirect header inclusion. Our challenge thus is: how can we close that door again?

A first, obvious idea would be to simply move all definitions to a source file, leaving only non-defining declarations in a header file. But, unfortunately, that idea only works for functions and variables; it does not work for classes, enumeration types, type aliases, templates, and so on. For all these other entities, a definition—one and only one—needs to be present (or at least visible or reachable: see Chapter 11) in every translation unit that uses these entities. For this reason, you cannot move the definitions of `square()` (a template) or `Oddity` (a type definition) out of the `BadMath.h` header file of `ExA_10`, for instance, and into some source file. This is entirely analogous to why in `Ex11_01` you could not move these same definitions out of the interface of the `math` module, and into a module implementation file.

So, what do we have so far? On the one hand, each header file unavoidably gets included more than once into some translation unit, eventually. Which, in combination with the basic ODR rules, suggests that you should never define any entities in a header. Yet, on the other hand, when you define a type or template you of course often want to use it in more than one translation unit. While defining Box classes over and over again has been barrels of fun, you normally want to do define a given class only once, and then reuse that same definition throughout your program. Which, in non-modular C++, means you have no choice *but* to define it in a header file, to then include it where needed.

In standard, idiomatic C++, the solution to this apparent stalemate is to surround each and every header file with a so-called *#include guard*. This pattern is based on a clever use of the #ifndef and #define preprocessing directives. You have already seen that you don't have to specify a value when you define a macro:

```
#define MY_IDENTIFIER
```

This creates MY_IDENTIFIER, so it exists from here on and represents an empty character sequence. You also know that you can use the #ifndef directive (short for #if !defined) to test whether a given identifier has been defined and include code or not in the file depending on the result:

```
#ifndef MY_IDENTIFIER
  // The code down to #endif will be placed in the source file if MY_IDENTIFIER
  // has NOT been defined. Otherwise, the code will be omitted.
#endif
```

All the lines following #ifndef down to the #endif directive will be omitted entirely from the file if the identifier, MY_IDENTIFIER, has been defined previously. It will no longer be considered by the preprocessor, and thus of course not by the compiler either. If MY_IDENTIFIER is defined, it is as if that entire block of code were never there. And so, more to the point: if MY_IDENTIFIER is defined, it is as if any potentially duplicate entity definitions in that conditional code block were never there.

These observations form the basis for the #include guards that you can use to ensure that the contents of header files are never duplicated in a translation unit. Applied to the BadMath.h header from before, this pattern looks as follows:

```
// BetterMath.h - A second, better attempt at creating a header file
#ifndef BETTER_MATH_H
#define BETTER_MATH_H

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

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

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

#endif  // The end of the #include guard
```

Naturally, you can choose any unique identifier instead of BETTER_MATH_H, as long as you use a different identifier in each header. Different naming conventions are used, although most base these names on the name of the header file itself. In this appendix we'll use identifiers of the form HEADER_NAME_H.

To show the effects of an #include guard, we now include BetterMath.h twice in the same translation unit and walk you through what happens.

```
// ExA_10A.cpp
// Including the same header twice in the same translation unit without violating ODR
import <iostream>;
#include "BetterMath.h"
#include "BetterMath.h"

int main()
{
  std::cout << square(1.234) << std::endl;
}
```

The first #include directive will include the various C++ entity definitions in BetterMath.h because at that point BETTER_MATH_H is not yet defined. In the process, though, the preprocessor defines the macro BETTER_MATH_H. Any subsequent #include directive for BetterMath.h in the same translation unit will thus no longer include any code at all because BETTER_MATH_H will have been defined previously. It is as if that second #include directive was never there at all. Have another look at the #include guard in BetterMath.h, and we're sure you'll appreciate the cleverness of this trick.

■ **Tip** Surround all your headers with an #include guard to eliminate the potential for violating the one definition rule when that header is included twice into the same translation unit.

■ **Tip** Most compilers offer a #pragma preprocessing directive to achieve the same effect as the relatively verbose #include guard pattern. Simply placing a line containing #pragma once at the beginning of a header file generally suffices to prevent duplication of the header's contents as well. While nearly all compilers support this #pragma, it is not standard C++, so for this appendix we'll continue to use #include guards in header files.

Defining Functions and Variables in Headers

While #include guards prevent duplicate definitions *within a single translation unit*, they do not prevent violations of the program-wide ODR rules when you include the same variable or function definition into *multiple translation units of the same program*. Suppose you have a program that, next to the BetterMath.h header of ExA_10A, consists of the following five files:

```
// Hypot.h - Declaration of an external math::hypot() function
#ifndef HYPOT_H
#define HYPOT_H

namespace math
{
  // Computes the length of the hypotenuse of a right-angle triangle
  double hypot(double x, double y);
}

#endif
```

```
// Hypot.cpp - Definition of the math::hypot() function
#include "BetterMath.h"  // For the square() function template definition
#include <cmath>         // For std::sqrt()

// Caution: See Chapter 11 for why you should always use std::hypot()
// over this naïve definition (based on the Pythagorean Theorem)!
double math::hypot(double x, double y) { return std::sqrt(square(x) + square(y)); }

// Pow4.h - Declaration of a math::pow4() function
#ifndef POW4_H
#define POW4_H

namespace math
{
  // Same as std::pow(x, 4)
  double pow4(double x);
}

#endif

// Pow4.cpp - Definition of the math::pow4() function
#include "BetterMath.h"  // For the square() function template definition²

namespace math
{
  double pow4(double x) { return square(square(x)); }
}

// ExA_11.cpp
import <iostream>;
#include "Hypot.h"
#include "Pow4.h"

int main()
{
   std::cout << math::hypot(3, 4) << '\t' << math::pow4(5) << std::endl;
}
```

The Hypot.cpp and Pow4.cpp source files define hypot() and pow4(), respectively. Both are functions in the math namespace, as declared in the corresponding header files. Refer to Chapter 11 for the different ways you can define entities within a namespace after declaring them first in an interface file. Whether this interface file is the module's interface file or an included header makes no difference for this.

The main point that we want to make with ExA_11, though, is this: Because Hypot.cpp and Pow4.cpp both contain an #include directive for BetterMath.h, both their translation units contain a definition for the isOdd() and getOddity() functions. This violates the program-wide ODR rule that stipulates that only one definition of the same function is allowed in the entire program. And since a C++ preprocessor considers each translation unit independently, no amount of #include guards or #pragmas can protect you from this.

■ **Note** The `#include` guards surrounding `Hypot.h` and `Pow4.h` are not really required, because neither header contains a definition. It is common practice, though, to add an `#include` guard (and/or a `#pragma once` directive) to all your headers, even to those where it is strictly speaking not required. The reason is because that guideline is both easy to remember and safe.

■ **Note** Recall that the program-wide ODR rules only restrict the definitions of functions and non-`const` variables. In `ExA_11`, for instance, only the definitions of `isOdd()` and `getOddity()` therefore violate ODR. Multiple definitions for the same type or template (such as the `enum class Oddity` or the `square()` template) are allowed by the ODR rules (and often even required), as long as they are all identical and there is at most one such definition per translation unit. The reason that the `lambda` variable does not violate ODR in `ExA_11` is because it is qualified as `const`, which gives its name internal linkage in both translation units in which it is included. The resulting definitions for `lambda` are therefore not seen as two definitions of the same external entity, but rather as independent definitions of two different constants, each internal to its respective translation unit.

Does this mean that you can never define functions and non-`const` global variables in a header? That you should always define functions and variables in a source file instead? Of course not. And we already gave away the answer in the section on ODR: the program-wide ODR rules only apply to regular functions and regular variables; they do not apply to *inline functions* and *inline variables*. For inline entities, multiple definitions are allowed within a program, as long as they are all identical, and no two definitions appear within the same translation unit—just like for type and class definitions.

But before we show you how you can create inline definitions in header files, perhaps first a brief discussion on when and why you would want to place certain definitions in a header file in the first place—besides to avoid having to type the same declarations twice, which of course is already a perfectly understandable sentiment in its own right.

Inlining

Defining a function in a header file may result in better runtime performance. Roughly speaking, a function call involves pushing a record on the call stack, getting things ready such that the function body has access to the values of all arguments, jumping to the location in memory where the compiled code for the function body is located, evaluating the function's statements, popping the record of the call stack again, and in doing so somehow retrieving the function's return value (if any). So quite a lot of work. If the compiler has access to the function definition, however, it can simply copy-paste the code from the function body into the calling function and avoid all these extra steps. That is, the compiler can then for instance rewrite an expression such as `isOdd(something())` into `(something() % 2 != 0)`, and by doing so, avoid the (relatively speaking) far more expensive function call of `isOdd()`. This compiler optimization is called *inlining*.

■ **Tip** For most functions, the overhead of a function call is either negligible compared to the time it takes to execute the function body, and/or they are never called often enough for inlining to make any difference in the overall performance of the program. You should therefore only consider facilitating compilation-time inlining of function calls for functions that are both small (as in: consist of only one or a few statements), and are likely to be called many times in time-critical code. Prime examples where inlining can make a difference include basic primitives such as `std::abs()` or our own `isOdd()`, and function members such as `size()` or `operator[]()` for containers (see Chapter 20). These functions are clearly not only very small, they are also very likely to be used inside loops that process a lot of data. In general, though, our advice is to not worry about optimizations such as inlining unless performance profiling tells you to!

Inline Functions and Variables in Headers

To define an inline function or variable, all you have to do is add the `inline` keyword in front of the definition. Here is how you do this in our running example.

```
// ProperMath.h - Your first, proper header file
#ifndef PROPER_MATH_H
#define PROPER_MATH_H

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

const inline double lambda{ 1.303577269034296391257 };       // Conway's constant

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

#endif
```

This header, `ProperMath.h`, is finally a fully ODR-safe equivalent of the `math` module of `Ex11_01` (you can find this header in `ExA_12`). The `#include` guard prevents ODR violations when you (indirectly) include the header multiple times into the same single translation unit, and by turning `isOdd()` and `getOddity()` into `inline` functions, you have ensured that you can safely include `ProperMath.h` not just in one translation unit, but in as many different translation units of the same program as you want.

Implicitly instantiated instances of a function template (see Chapter 10) are inline by default. Even though you are allowed to explicitly add `inline` in front of the definition of `square()` in `ProperMath.h` if you want to, there is no real need to do this.

Whether you add `inline` to constants such as `lambda` is not that important either.[7] As argued earlier, without the `inline` specifier that we added to the definition of `lambda` in `ProperMath.h`, you simply end up with a local copy of this variable in each translation unit in which it is included. The reason is that the names of non-inline, `const` variables have internal linkage by default. Take care, though: had the `lambda` variable not been `const`, the `inline` specifier would no longer have been optional, because then `lambda` would've had external linkage!

The order in which you specify `const` and `inline` in front of a variable definition does not matter.

[7] The only noticeable difference between an inline constant and a non-inline constant at the global or namespace scope occurs when you take the address of this constant in different translation units, and then compare these. In the former case, there is only one (external) variable, which means it has the same address in all compilation units; in the latter case, there is one (internal) variable per translation unit, which means the address will be different in each compilation unit.

■ **Note** The `inline` keyword is merely a suggestion to the compiler to inline a function, nothing more. If the compiler concludes that it would be more efficient not to inline a particular call, it will still ignore your suggestion and generate a function call instead. Conversely, if it believes this to be more efficient, the compiler will happily inline any non-inline function definition that is present in the same translation unit as well.

■ **Caution** Defining functions and variables in a header file comes at a cost, especially during development. Each time you change a definition in a header file, all translation units in which that header is included will need to be recompiled. If you change a definition in a source file, on the other hand, only that source file needs to be recompiled. (In both cases, the program of course has to be linked again.) Because function definitions change far more often than their prototypes, defining functions in source files rather than in header files generally speeds up development tremendously.

Modules do not suffer from this last issue nearly as much as headers do, so the considerations are different. We discuss this further in the next section.

Inline Functions and Variables in Modules

As you know, modules are designed to be self-contained, independent, composable units (see Chapter 11). One of the key goals in designing the modules language feature was that modules should reduce compilation times, both when creating an executable from scratch, and when re-creating one after some changes during development. The latter is often referred to as *incremental compilation*. These concerns led to the restriction that, as long as the interface of a module does not change, you should be able to work inside that module and recompile its files as often as you want, without ever having to recompile any of the external translation units that import it. Only if you change the exported interface of a module should you have to rebuild its binary interface file, and then recompile all translation units that consume this interface.

As a consequence, your compiler is by default not allowed to inline a call to a function that is imported from a different module. By default, a function's definition is not part of a module's interface, not even if that definition appears in a module interface file. The definition of an exported function only becomes part of the module interface if you explicitly mark it as `inline`, and only then is the compiler allowed to inline this definition in consuming translation units[8].

Mind you, including a function's definition in the module interface does imply that any consequent change to its function body necessitates recompilation of consuming translation units, so even in modular code you should still only consider using `inline` for functions that are small, whose definition is unlikely to change, and that are likely to be called a significant amount of times in time-critical code.

Defining Classes in Headers

So far, we mostly focused on definitions of functions and variables in this appendix, mostly because they are compact. Of course, non-modular C++ code contains class definitions at least as often. In this section, we briefly explore how header and source files that define classes and their members typically look.

[8] Compiler vendors will likely add a switch to allow you to overrule this default behavior and give the compiler the green light to inline any exported functions it deems fit, regardless of whether you specified them as inline. The idea is then that you would rely on the default behavior during development to reduce recompilation times, but use that compiler switch for the final, fully optimized binary that is intended for your users.

Because you need access to the definition of a class in every translation unit in which objects of that class are used, classes themselves are generally defined in header files. Take this header, for instance, which contains a definition of an old-time favorite: the Box class (see Chapter 12 and beyond).

```
// Box.h
// Your first header file with a class definition
#ifndef BOX_H
#define BOX_H

class Box
{
public:
  Box();
  Box(double length, double width, double height);

  double getLength() const;
  double getWidth() const;
  double getHeight() const;

  double volume() const;

private:
  double m_length;
  double m_width;
  double m_height;
};

#endif
```

Because member definitions have to obey the same ODR rules as regular functions and variables, we kept things safe and easy in this first version of the header and defined all member functions in a source file. Besides the #include directive at the top, and the lack of a module declaration, this source file looks precisely like the analogous module implementation files (see Chapter 12).

```
// Box.cpp
#include "Box.h"

Box::Box() : Box(1.0, 1.0, 1.0) {}
Box::Box(double length, double width, double height)
  : m_length(length), m_width(width), m_height(height)
{}

double Box::getLength() const { return m_length; }
double Box::getWidth() const  { return m_width; }
double Box::getHeight() const { return m_height; }

double Box::volume() const
{
  return m_length * m_width * m_height;
}
```

Here is a small test program to show you that this non-modular separation of interface and definition works.

```
// ExA_13.cpp - Including a class definition from a header
import <iostream>;
#include "Box.h"

int main()
{
  Box boxy{ 1, 2, 3 };
  std::cout << boxy.volume() << std::endl;
}
```

Note that this `main()` function is again able to call the `volume()` member function, even though no definition of this member is present in its translation unit. The reason that this works, of course, is again because the compiler simply flags this call as an external function call, after which the linker matches it to the correct definition in the `Box` translation unit. You should remove the definition of `volume()` from `Box.cpp` to see what errors the linker then produces.

Inline Class Members

You can also define class member functions directly in the header file, but only if you use inline definitions. To illustrate your options, you can remove the `Box.cpp` source file of `ExA_13` and replace the `Box.h` header file with this version.

```
// Box.h
// Inline class member definitions
#ifndef BOX_H
#define BOX_H

#include <ostream>   // Caution: do not use import <ostream>; here!

class Box
{
public:
  Box() = default;   // In-class definition, and thus implicitly inline
  Box(double length, double width, double height);

  // In-class member definitions are implicitly inline
  double getLength() const { return m_length; };
  double getWidth() const  { return m_width; };
  double getHeight() const { return m_height; };

  double volume() const;

private:
  double m_length {1.0};
  double m_width {1.0};
  double m_height {1.0};
};
```

```
// Out-of-class member definitions must be explicitly marked as inline
inline Box::Box(double length, double width, double height)
  : m_length(length), m_width(width), m_height(height)
{}

inline double Box::volume() const
{
  return m_length * m_width * m_height;
}

// Definitions of non-member functions of course must be inline as well
inline std::ostream& operator<<(std::ostream& out, const Box& box)
{
  return out << "Box(" << box.getLength() << ", "
             << box.getWidth() << ", " << box.getHeight() << ')';
}

#endif // End of the #include guard
```

In the global module (see Chapter 11), all in-class member definitions are implicitly inline. This is why `inline` is not needed in front of the defaulted default constructor or the three getters of `Box` in `Box.h` (you can add `inline` there, if you want to, but you do not have to). When defining a member out-of-class, however, its definition is inline only if you explicitly mark it as such. Without the `inline` specifier in front of the definition of either the non-default constructor or `Box::volume()`, ODR would thus prohibit you from including `Box.h` in more than one translation unit of the same program (perhaps give that a try as well?).

Note that we also added a non-member operator overload for the stream output operator, `<<`, to `Box.h` (operator overloading is discussed in Chapter 13). Because this definition is inside a header file, you of course have to add the `inline` specifier there as well. Notice also that we used an `#include` directive rather than an `import` declaration to gain access to the definition of the `std::ostream` class in `Box.h`. In fact, you are not even allowed to use an `import` declaration here. And here is why:

■ **Caution** Never add module `import` declarations to a header file. By design, build systems should be able to discover module dependencies without full preprocessing. This implies that `import` declarations are not allowed to originate from either an `#include` directive or a macro expansion. This also relates to why all `import` declarations have to appear in a module's purview before any other declarations (as seen in Chapter 11). All these limitations are put in place specifically to facilitate efficient dependency discovery, which is something build systems need to do when deciding in which order they should compile a larger number of program files.

Modules vs. Headers

We already covered most of the advantages that modules offer over headers, either in Chapter 11 or in this appendix. We start this final section by briefly recapitulating these.

- Modules result in *reduced compilation times* because every module file is processed by the compiler only once. Header files, on the other hand, are included and processed over and over again in every translation unit that consumes them, directly or indirectly.

- Modules result in *faster incremental recompilation* as well, because only changes to the module's interface trigger recompilations of consuming translation units. Changes to the bodies of non-inline functions, or changes to local entities, require only the module file itself to be recompiled.
- Modules are *self-isolating*, in the sense that only changes to a module's interface can ever impact the compilation of other translation units. Header files often leak declarations and definitions of local entities into the including translation units, as well as `#include` directives for lower-level headers, or dangerous macro definitions—all of which may lead to inadvertent name clashes or other compilation issues. Just think of the havoc that the `min()` and `max()` macros defined by the Windows API headers often wreak in C++ code!
- The order in which you import modules never matters, where with headers it could. The reason is that a module interface is never impacted by any macro definitions in the importing translation unit, nor does it ever inject any new ones.
- With modules, you can very easily and naturally *define all entities directly in one file*. There is no need for constructs such as `#include` guards, `#pragmas`, unnamed namespaces, or `inline` specifiers—at least not to avoid ODR violations (you may occasionally still want to add `inline` to improve performance, though, for instance). Mind you, there can still be value in separating a module's interface from its implementation. This may allow you to present a more compact, cleaner overview of the interface, for instance, or to hide confidential implementation details when shipping libraries to customers. But at least with modules, when you do create multiple files, it will be for the right reasons, and not because of a fear of ODR violations or of increased recompilation times.

All this combined should make one thing crystal clear: if your compiler and build system support it, you should *always* use modules to organize new code. No question about it. Yet of course there is decades' worth of non-modular code out there. Does this mean that you have to rewrite this all to benefit from these advantages? Not quite. C++20 also supports the notion of *header units*, which allow you to treat most header files as module interface files without reworking these header files.

Header Units

In the section on organizing non-modular code, we turned a module interface file into a header file, side-stepping the various ODR issues that this brings as we went along. But of course you are normally faced with the opposite task: turning non-modular code into modules to leverage the advantages that we listed earlier. As an example, we will start again with `ProperMath.h` from `ExA_12`.

```
// ProperMath.h - Your first, proper header file
#ifndef PROPER_MATH_H
#define PROPER_MATH_H

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

const inline double lambda{ 1.303577269034296391257 };  // Conway's constant
```

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

#endif
```

Your first option, of course, is to simply plow on, remove the `#include` guards, add the `module` declaration, add `export` keywords where desired, and rename the header file. If the header file is paired with a source file, you of course have to add a matching `module` declaration there as well. On the whole, this is a fairly straightforward process for most headers.

While rewriting your code like that is certainly a viable option—especially in the longer term—it does still takes some effort. You may not always have the time to convert all code at once. Also, sometimes the same source code may still need to work for other programs in which you cannot yet use modules—for instance, because these program use an older compiler to target different platforms. Or perhaps the header files in question are part of a third-party library out of your control.

In cases like that, there is also a second option. Clearly the steps required to turn non-modular program files into module files are fairly predictable and repetitive, at least for most headers (and source files). And if there is one thing that computers still do better than humans, it's predictable, repetitive jobs. The language therefore allows you to compile and subsequently import most header files as if they were interface files of a module. The general syntax you use to import such a synthesized module looks as follows.

```
import "some/local_path/local_header.h";
import <some/external_path/external_header.h>;
```

That is, instead of a module name (see Chapter 11), you can also specify the name (or, in general, the relative path) of a header file in an `import` declaration, surrounded with either double quotes or angle brackets. The way the compiler resolves these path names will of course be analogous to the way the preprocessor resolves those of `#include` directives (see earlier).

An `import` declaration that names a header file acts as if it imports some compiler-generated module, whose interface is extracted from the header file. The synthesized translation unit that specifies this interface is referred to as a *header unit*. In this header unit, all declarations within the original header file become exported. The only difference between the two following import declarations is therefore that the former makes an `isOdd()` function available to the importing unit, whereas the latter does not. Of course, this difference only exists because in the `math` module of `Ex11_1` we choose not to export the module-local `isOdd()` function.

```
import "ProperMath.h"; // Imports a module whose interface is synthesized from ProperMath.h
import math;           // Imports the hand-written math module of Ex11_1
```

The exact steps you need to compile header units are vendor specific, so you should consult your compiler's documentation for this.

Only so-called *importable* headers can be compiled and subsequently imported using this syntax. What precisely constitutes an importable header is again implementation defined. Likely, though, examples of headers that are not importable will include those that define certain types of preprocessing macros.

Even headers that define basic macros can be importable, though. The reason is that, unlike regular modules, header units do export all preprocessing macro definitions that are defined at the end of the preprocessing phase. Should `Windows.h` for instance be importable (which is somewhat doubtful), then an `import <Windows.h>;` declaration will by default import the `min()` and `max()` macros.

■ **Caution** Header units are not impacted by any macros that are defined at the point where import declaration occurs. That is, defining a `NOMINMAX` macro prior to a hypothetical `import <Windows.h>;` declaration, for instance, would not work. Like with all module interface units, a header unit is only compiled once, so any macros that you need to configure this unit should be defined at the moment that it's synthesized, not when you import it. One option is to define these by passing the necessary flags to your compiler when synthesizing the header unit (consult your compiler documentation).

As mentioned in Chapter 2, all headers of the C++ Standard Library, except for those headers that originate from the C Standard Library (such as `<cmath>`, `<cassert>`, and so on), are guaranteed to be importable. We have exploited this fact consistently throughout the book.

■ **Tip** Most compilers have an option to implicitly treat certain `#include <header>` directives as if they were `import <header>;` declarations. This option typically exists at least for all Standard Library headers, but possibly also for your own headers or those of other dependencies. As this can significantly speed up translation, without reworking any code at all, this may be worth looking into.

Summary

This appendix discussed various topics, most of which are particularly relevant in non-modular code. The important points from this chapter include the following:

- The preprocessing phase executes directives to transform the source code in a translation unit prior to compilation. When all directives have been processed, the translation unit will only contain C++ code, with no preprocessing directives remaining.
- You can use the `#define` directive to define object- or function-like text replacement macros. In C++, you should mostly use variables or function templates instead, though.
- You can use conditional preprocessing directives to control which blocks of code make it into a translation unit. This is particularly convenient if the same program files should compile with different compilers, or for different target platforms.
- You insert the entire contents of any file into another file using an `#include` directive. The files that you include this way will nearly always be header files.
- The `assert()` macro enables you to test logical conditions during execution and issue a message and abort the program if the logical condition is `false`. You can use `static_assert()` to similarly check conditions at compile time.
- Each entity in a translation unit must have only one definition. If multiple definitions are allowed throughout a program, they must still all be identical. Multiple definitions are not allowed for functions, member functions, or global, non-`const` variables, though, unless they are specified to be `inline`.
- A name can have internal linkage, meaning that the name is accessible throughout a translation unit; external linkage, meaning that the name is accessible from any

translation unit; module linkage, meaning that the name is accessible from any translation unit of the same named module; or it can have no linkage, meaning that the name is accessible only in the block in which it is defined.

- You use header files to contain definitions and declarations required by other headers or your source files. A header file can contain template and type definitions, enumerations, constants, function declarations, `inline` function and variable definitions, and named namespaces. By convention, header filenames use the extension `.h`.
- Your source files will typically contain the definitions for all non-`inline` functions and variables declared in the corresponding header. This also typically includes the definitions of class members.
- You can use conditional preprocessing directives in the form of an `#include` guard to ensure that the contents of a header file are never duplicated within a translation unit. Most compilers also support the more convenient `#pragma once` directive for this same purpose.
- In new code you should always use modules, and existing code should gradually be made modular.
- Header units, which are module interface units that the compiler generates based on a legacy header file, can help leverage the advantages of modules without reworking existing code.

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, the solutions are available in the same repo you downloaded this appendix from, but that really should be a last resort.

Exercise A-1. Define your own `ASSERT()` macro that, like a built-in `static_assert()` declaration optionally does, accepts two arguments: a condition and a message. If the given condition evaluates to `false`, it should print the given message to `std::cerr` and then call `std::abort()`, a function declared in the `<cstdlib>` header. Define your function-like macro in an `ASSERT.h` header file and use it to replace the assertion in `ExA_03.cpp`.

Exercise A-2. Make the `ASSERT()` macro in Exercise A-1 do nothing (not even evaluate the given expression) if the object-like `NDEBUG` macro is not defined. Define `NDEBUG` to test that it works.

Exercise A-3. Create an exception class derived from `std::logic_error` that, next to the inherited `what()` function, adds a `where()` function that returns a reference to a `std::source_location`. This `source_location` facilitates pinpointing where exactly the exception was thrown. Make sure that users of the class can, but do not have to, explicitly create the `source_location` object themselves. In the spirit of this chapter you should of course define the new class in a header file and its members in a source file. Write a small test program to show that it works.

Exercise A-4. In the solution of Exercise A-3, move all member definitions from the source file to the header file. For the class's constructors, you should use in-class definitions; for its `where()` function, use an out-of-class definition. Make sure, of course, that at least in principle the header could be included in any number of translation units.

Exercise A-5. Write a program that calls two functions, `print_this(std::string_view)` and `print_that(std::string_view)`, each of which calls a third function, `print(std::string_view)`, to print the string that is passed to it. Define each function and `main()` in separate source files and create header files to contain the prototypes for `print_this()` and `print_that()`. For the sake of the exercise, don't put the prototype of `print()` in its own header (although one normally would, to avoid having to duplicate the function's prototype).

Exercise A-6. Modify the program in Exercise A-5 so that the definition of `print()` is inside a header file that is included in the source files of both `print_this()` and `print_that()`.

Exercise A-7. Modify the program in Exercise A-6 so that `print()` uses a global integer variable to count the number of times it has been called. Output the value of this variable in `main()` after calls to `print_this()` and `print_that()`. Then do the same with the program in Exercise A-6, and work around any issues you may encounter without moving the definition of `print()`.