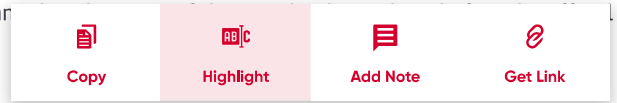
Chapter 9. Modules and Concepts

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can release of these titles.



This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *learnmodcppfinance@gmail.com*.

Introduction

One of the most anticipated additions to C++20, *modules*, received a lot of fanfare at C++ conferences and in the blogosphere leading up release of the updated Standard. Among other features, modules offer potentially very significant advantages over the currently employed header files with `#include` pre-processor statements, as will be discussed below.

It was hoped that before the time this book went to press, modules would have become a common fixture in newer development projects. Unfortunately, this has not been the case, although considering this is a major change in the C++ language, it is somewhat understandable for compiler implementations to take time, as well as for adoption by developers. Visual Studio provided working support for modules by around mid-2021, but implementation delays in Clang and gcc continued into 2023. There has also been some debate even on whether there should be a new set of preferred file extensions, and if so, what they should be **{1}**.

Still, as "C++ Modules were designed to bring more safety to your programs, while dramatically reducing compile time, resulting in overall increased productivity" **{2}**, it is probably a good idea to start getting some familiarity with how modules work.

Concepts were also an eagerly awaited feature in C++ that give programmers more control over template programming. Invalid function inputs can be pinpointed at compile time without being buried within the

avalanche of superfluous error messages associated with templates that can obscure the location of the offending code. A simple example is attempting to use a `string` as a template parameter in a function that contains a multiplication operation, but in real-world programming where we need to solve more interesting and complex problems, sorting through what can be a massive volume of compiler output can make debugging a nightmare.

Concepts can also indicate the intentions behind a function or class template. This should provide better clarity to programming teams that responsible for maintaining a production level code base.

All three major compilers fully support concepts.

Modules

To start with an example, a module can be defined in a single file, containing both declarations and implementations, although as we will see later, one also has the option to utilize a separate implementation file. To begin, our old friend `SimpleClass` is implemented inside a module called `SingleFileExample`. More specifically, this defines the *primary module interface unit* for this module, as indicated by the first line of code, containing the `export` keyword (1):

```
export module SingleFileExample;          // (1)

// SimpleClass declaration and implementation together
export class SimpleClass                  // (2)
{
public:
   SimpleClass(int k) :k_{ k } {}

   int get_val() const
   {
      return k_;
   }

   void reset_val(int k)
   {
      k_ = k;
   }

private:
   int k_;
};
```

The `SimpleClass` definition is also preceded by the `export` keyword. This makes it available to any code that *imports* the module. For example, to use this class in a `main()` function residing in a separate file, one would use the `import` keyword at the top with the name of the module, as shown here:

```
import SingleFileModule;

. . .

int main()
{
      SimpleClass sc{ 10 };          // Exported from SingleFileExample module
      . . .
}
```

The `import` keyword might remind you of using `#include` for a header file, but there are important differences (to be discussed momentarily). Also, had `export` not preceded the `SimpleClass` class definition, this code would not compile, as the class would not have been visible within the `main()` function.

Similarly, modules can contain standalone non-member functions. We can use an example here to also demonstrate a non-exported function.

```
export module SingleFileExample;

import <vector>;                // Standard Library Header Units are imported...
import <algorithm>;                  // to be discussed in the next section.
import <ranges>;
import <iterator>;        // std::back_inserter(.)

. . .

std::vector<double> vector_fcn_helper(const std::vector<double>& x)
{
   std::vector<double> y;
   y.reserve(x.size());
   std::ranges::transform(x, std::back_inserter(y),
            [](double q) { return 3 * q; });

   return y;
}

export std::vector<double> vector_fcn(const std::vector<double>& x)
{
   return vector_fcn_helper(x);
}
```

In this case, `vector_fcn_helper(.)` is not exported, but it is called from the exported `vector_fcn(.)`. In this sense, the latter serves as an interface, while the former holds an internal implementation. If we attempted to write:

```
int main()
{
      . . .
      std::vector v{ 1.0, 2.0, 3.0 };
      std::vector w = vector_fcn_helper(v);          // Will not compile!
}
```

the code would not compile. Instead, we are forced to use the exported interface function:

```
int main()
{
      . . .
      std::vector v{ 1.0, 2.0, 3.0 };
      std::vector w = vector_fcn(v);
}
```

You might also have noticed we used `import` for the Standard Library headers rather than using `#include` just after the module definition begins. Salient differences with `import`, including Standard Library headers, will be discussed next.

NOTE

Although there are no standard file extensions in C++, for years there have been generally accepted extension configurations for header files—eg `.h`, `.hpp`, `.hxx` etc—and `.cpp`, `.cxx` etc for implementation source files. When modules first started hitting the scene, however, there was no such consensus for module interface files. In Visual Studio, the default property was an `.ixx` extension, while for Clang it was `cppm`, which arguably was more descriptive as it could be interpreted as the common `.cpp` plus `m` for module. Meanwhile, the gcc compiler default was to just use `.cpp` for both traditional implementation files and module interfaces.

Recent versions of Visual Studio 2022 (at least as of v 4.8), however, now also recognize `.cppm` in its default properties, so perhaps there is some coalescing around this extension format. This extension is used for module interface files in the sample code for this chapter.

Standard Library Header Units

Proposals to the ISO C++ Committee for reorganizing the Standard Library into Standard Modules **{3}** were also drafted and submitted for inclusion in C++20, but this effort was deferred until C++23 **{4}**, and implementations for each of the three mainstream compilers are still not yet quite complete as of early 2024. Once available, one will be able to just write:

```
import std;
```

and be done. A very simple example would be:

```
export module Test;

import std;

export void test_vector()
{
      std::vector<int> v{1, 2, 3};
      for (int x : v)
      {
            std::cout << x << " ";
      }

      std::cout << "\n\n";
}
```

In the interim, as a placeholder, Standard Library *header units* that guarantee "[e]xisting `#include` (s) of standard library headers transparently turn into module imports in C++20" can be used **{5}**. What this essentially means is preprocessor statements such as

```
#include <vector>
#include <algorithm>
```

can be replaced by importing their header unit equivalents:

```
import <vector>;
import <algorithm>;
```

Note that a semicolon is required at the end of each, as these are seen by the compiler as regular C++ statements, vs `#include` statements that are destined for the pre-processor.

This applies to almost all C++ Standard Library declaration files; however, due to complications arising in headers inherited from C -- eg, `<cassert>` and `<cmath>`, based on the legacy C headers `assert.h` and `math.h` respectively -- these are not covered and need to be handled with the usual `#include` pre-processor statements within what is called the *global fragment* of a module, indicated by a simple `module;` statement, which must be placed above `export module` statement as shown here. Then, if we add a function `math_fcn(.)` that requires `<cmath>` functions, our module would then take on something of the form:

```
module;
#include <cmath>       // Legacy C-derived headers go in the global fragment
                           // at the top of the file.
```

```
export module SingleFileExample;
import <vector>;         // Importing a Standard header unit
                                 // comes after the module name is defined.
. . .

export std::vector<double> math_fcn(const std::vector<double>& x)
{
   std::vector<double> y;
   y.reserve(x.size());
   std::ranges::transform(x, std::back_inserter(y),
            [](double q) {return std::sin(q) + std::cos(q); });

   return y;
}
```

Templates in Modules

Templates can also be written in module interface files, as shown in this example:

```
export module Templates;
import <iostream>;

export template<typename T>
T factored_polynomial(T a, T b, T c, T d)
{
      return (a + b) * (c + d);
}

export template<typename T>
void print(T t)
{
      std::cout << t << " ";
}
```

`import` vs `#include`

As modules have only barely gotten off the ground at this point, `#include`-ing traditional header files in the global fragment will just be a fact of life for a while, particularly with popular external libraries such as Boost, Eigen, Armadillo, etc:

```
module;
#include <boost/math/distributions.hpp>
#include <Eigen/Dense>

export module ModuleThatUsesMathLibraries;
```

```
import <random>;            // Standard Library Header Unit

...
```

When writing new code however, when possible, preferring modules over header files can provide some distinct advantages. The first is the fact that a module will not "leak" imported modules within itself when imported elsewhere. That is, if a module `A` imports another module `B`,

```
// Define module A that imports module B:
export module A;
import B;
```

then if `A` is imported into another module `C`, `B` will not be imported as well unless it also is explicitly instructed to do so. If not, attempts to use exported functions in module `B` inside of module `C` will fail to compile.

```
// Define module C that imports module A:
export module C;
import A;
import B;           // Must be explicitly imported if functions
                    // in B are also to be used inside module C
```

This is in contrast with `#include` (d) header files that will leak anything it itself has `#include` (d). For example, suppose a header file `MyHeader.h` includes another user-defined header file `YourHeader.h`, and the STL `<vector>` header:

```
// MyHeader.h
#include "YourHeader.h"
#include <vector>

...
```

If `MyHeader.h` is `#include` (d), say again in the translation unit containing `main()`, then it will also carry with it functions in `YourHeader.h` and the `std::vector` class, such that the following will compile:

```
#include "MyHeader.h"
int main()
{
    // This will compile:
    auto y = my_header_fcn(...);

    // But so will these lines:
    auto z = your_header_fcn(...);
```

```
    std::vector <double> v;
}
```

In realistic situations where many more header files might be involved, losing track of what is included and what is not can potentially lead to unpleasant surprises at compile time and/or runtime. With modules, the programmer has greater control over what is imported.

On the flip side, you might *want* to export a module imported by another module. For this, you need to use an `export import` statement. For example, suppose we have the following module:

```
export module VandelayIndustries;

export class Latex
{
public:
    Latex(int quantity, double unit_price) :
        quantity_{ quantity }, unit_price_{ unit_price } {}

    double total_sale_value() const
    {
        return quantity_ * unit_price_;
    }

private:
    int quantity_;
    double unit_price_;
};
```

And then, suppose we also have a module `ExportImportBusiness`, but we want users who import this module to also have access to the `Latex` class. This is accomplished by using `export import`:

```
export module ExportImportBusiness;
export import VandelayIndustries;
import <string>;
import <iostream>;
import <format>;

export void order(const std::string& order_code, const Latex& latex)
{
    std::cout
        << std::format("Order: {}, Total Price: {}", order_code, latex.total_sale_value()) << "\n";
}
```

Then, in a separate translation unit, importing the `ExportImport` module will also allow us to use the `Latex` class without explicitly importing `VandelayIndustries`:

```
import ExportImport;

void export_import_example()
{
      Latex latex{ 2, 10.50 };            // exported from ExportImport module
      string order_code{ "X106224" };
      order(order_code, latex);
}
```

The upshot is now you can control which modules are exported from another module, rather than having them leak by default.

Two final remarks are as follows. First, these same safeguards will also be in effect if a module is imported into non-module code, such as the earlier example where the `SingleFileExample` module was imported into the translation unit containing `main()`. On the other hand, second, header files that are `#include`(d) in the global fragment of a module *will* leak to any destination where the same module is imported.

Declarations in Module Interfaces

During the pre-processor phase of the build process, with each time a traditional header file is `#include`(d) in a translation unit, its declarations are reparsed. As the same header file can be included in multiple translation units, it can itself include other header files (which are leaked). In a real world financial library or system there could be hundreds of different header files in play, plus those included from external libraries, so due to basic combinatorics these redundancies will add up and slow down the build process.

With modules, these redundancies are eliminated, as declarations in a module interface are precompiled into a single file that is reused, eliminating the repeated reparsing that takes place with traditional header files. As a result, using modules can potentially reduce build times. Similar results could also be obtained with traditional build methods by using precompiled headers; however, this is usually platform-dependent and hence not standard, while modules are.

Technically, these reductions in build times are tempered by the fact that modules can reduce parallelism of a build as compared to the traditional model, so at this stage, your mileage might vary. Given however that increasing efficiency in compilers is an active area of research, it might yield further breakthroughs down the road as the use of modules becomes mainstream, but this remains to be sorted out.

One definite positive consequence of precompiling declarations into a single file is it means we can say goodbye to “ugly” preprocessor directives **{6}**, such as include guards,

```
#ifndef MY_HEADER_H
#define MY_HEADER_H
```

```
// Declarations here . . .

#endif
```

that are at present commonly used to prevent *One Definition Rule* (ODR) violations **{7}**, or the nonstandard `#pragma once` that is sometimes used with compilers where it is defined.

NOTE

File extensions used for precompiled declarations that are generated with modules for the three major compilers are as follows.

- Visual Studio: `.ifc`
- Clang: `.pcm`
- gcc: `.gcm`

Separating Declarations from Implementation

In the opening examples, we utilized a coding style similar to what you would find in Java and C#, where a function or class declaration is combined with its implementation (although what happens behind the scenes is completely different). This can be seen quite clearly in the previous `SimpleClass` example.

From an organization point of view, it can be clearer to write the declarations separate from implementation, even though this still can be done in a single module interface file. Rewriting `SimpleClass` in this form, we would then have:

```
export class SimpleClass
{
public:
   SimpleClass(int k);
   int get_val() const;
   void reset_val(int k);

private:
   int k_;
};


SimpleClass::SimpleClass(int k) :k_{ k } {}

int SimpleClass::get_val() const
{
   return k_;
}
```

```
void SimpleClass::reset_val(int k)
{
   k_ = k;
}
```

In real life programming, of course, classes will not be this simple, so separating declarations from implementation can make the intent clearer and code maintenance easier.

One also has the option of splitting the declaration and implementation in a module into two separate files as is commonly done in the present. As your code increases in complexity, you will likely want to consider confining module interfaces to declarations alone, and write implementations in a *module implementation unit*, which can improve code maintenance further. One additional advantage to this separation is in cases of distributing compiled proprietary code while still providing the declarations, as is commonly done in the present by providing header files.

As an example, we could re-implement the `BlackScholes` class from Chapter 2 as follows. First, the primary module interface would house the class declaration alone.

```
// BlackScholesClass.cppm

export module BlackScholesClass;
import <array>;

export enum class PayoffType
{
      Call = 1,
      Put = -1
};

class BlackScholes
{
public:
      BlackScholes(double strike, double spot, double time_to_exp,
            PayoffType pot, double rate, double div = 0.0);

      double operator()(double vol);  // Mutable, for implied vol calculations (See Ch 2)

private:

      std::array<double, 2> compute_norm_args_(double vol);      // d1 and d2;

      double strike_, spot_, time_to_exp_;
      PayoffType pot_;
      double rate_, div_;
};
```

In this example, the enum class representing the payoff type and the `BlackScholes` class declaration are included in the module interface and exported.

The module implementation unit can then be written as shown here, with explanation to follow.

```
// BlackScholesClass.cpp          // (1)

module;                                 // Global fragment
#include <cmath>                  // (2)

module BlackScholesClass;       // (3)

import <numbers>;
import <algorithm>;

BlackScholes::BlackScholes(double strike, double spot, double time_to_exp,
      PayoffType pot, double rate, double div) :
      strike_{ strike }, spot_{ spot }, time_to_exp_{ time_to_exp },
            pot_{ pot }, rate_{ rate }, div_{ div } {}

double BlackScholes::operator()(double vol)
{
      using std::exp;
      const int phi = static_cast<int>(pot_);

      //double opt_price = 0.0;
      if (time_to_exp_ > 0.0)
      {
            auto norm_args = compute_norm_args_(vol);
            double d1 = norm_args[0];
            double d2 = norm_args[1];

            auto norm_cdf = [](double x) -> double
            {
                  return (1.0 + std::erf(x / std::numbers::sqrt2)) / 2.0;
            };

            double nd_1 = norm_cdf(phi * d1_);          // N(d1)
            double nd_2 = norm_cdf(phi * d2_);          // N(d2)
            double disc_fctr = exp(-rate_ * time_to_exp_);

            return phi * (spot_ * exp(-div_ * time_to_exp_) * nd_1 - disc_fctr * strike_ * nd_2);
      }
      else
      {
            return std::max(phi *  (spot_ - strike_), 0.0);   // std::max: <algorithm> is
                                                                                    // imported in the module interface.
      }
}
```

```
std::array<double, 2> BlackScholes::compute_norm_args_(double vol)
{
	double numer = log(spot_ / strike_) + (rate_ - div_ + 0.5 * vol * vol) * time_to_exp_;
	double d1 = numer / (vol * sqrt(time_to_exp_));
	double d2 = d1 - vol * sqrt(time_to_exp_);
	return std::array<double, 2>{ d1, d2 };
}
```

The first point (1) to note is module implementation files can just use the same extension as existing implementation files, eg `.cpp`.

The second point (2) is the inclusion of the `<cmath>` header is placed in the global fragment of the module *implementation* unit, as it is only needed here and not in the interface.

Finally (3), the same

`module BlackScholesClass`

statement is applied at the outset, indicating this implementation is part of the same module. This is what associates it with the module interface. Note that the `export` keyword is *not* used in the implementation unit. This can only be placed in a module interface unit.

This particular example hopefully provides a reasonable guide for writing module interfaces and implementations, but there are two general facts that should now be formally stated:

1. A module must contain one, and only one, primary module interface unit,
2. A module can contain more than one implementation unit.

Namespaces

Namespaces can also be enclosed within modules. There are two basic options: First, the entirety of a namespace can be exported:

```
export namespace OptionValuation
{
	enum class PayoffType
	{
		Call = 1,
		Put = -1
	};

	class BlackScholes
	{
	public:
```

```
        BlackScholes(double strike, double spot, double rate,
            double time_to_exp, PayoffType pot);

        double operator()(double vol);

    private:
        void compute_norm_args_(double vol);        // d_1 and d_2

        double strike_, spot_, rate_, time_to_exp_;
        PayoffType pot_;

        double d1_{ 0.0 }, d2_{ 0.0 };
    };
}
```

Or alternatively, individual classes and functions in a namespace can be exported, in the event there is an item in the namespace that you don't want to make available outside the module, eg:

```
namespace OptionValuation                   // export not applied to the entire namespace
{
    export enum class PayoffType   // export explicitly applied where required
    {
        Call = 1,
        Put = -1
    };

    export class BlackScholes
    {
        . . .
    };

    int mystery_function(int n);       // This function is not exported
}
```

Partitions

For modules with a larger scope, modules can be further divided into *partitions*. These can be useful for code organization and make it easier for different teams or developers to take responsibility for a particular section of the code. Implementation partitions can contain both declarations and implementations (this might seem a little confusing) that remain internal to the module, but they cannot export anything. Interface partitions *can* export functions, but ultimately they must be imported into the primary interface, and then exported to the outside world, using the `export import` designation.

(Maybe put a diagram here...or just refer to the reference in the next sentence and leave it at that.)

Comprehensive coverage of module partitions is a more advanced topic that would be beyond the intent of this chapter, but an extensive discussion of the topic can be found again in **{9}**.

Concepts

Concepts for templates were a long awaited and major enhancement in C++20. Just about anyone who has programmed with templates understands the agony of long and cryptic compiler error messages that obscure the reason that something went wrong, particularly in cases where a template parameter was invalid in a function template or class template.

For example, suppose we write the following function template:

```
template<typename T>
T add_them(const T& t1, const T& t2)
{
      return t1 + t2;
}
```

Then, because addition is defined for integers and real numbers, the following will compile no problem:

```
int sum_ints = add_them(531, 922);
double sum_reals = add_them(54.8, 18.4);
```

We could also use the function to add two Eigen library matrices, as vector addition is also defined by the `+` operator:

```
VectorXd xv(3);
VectorXd yv(3);

xv << 1.0, 2.0, 3.0;
yv << 4.0, 5.0, 6.0;

VectorXd sum_eig = add_them(xv, yv);
```

However, if we attempt to add two *Standard Library* vectors,

```
vector<double> u{ 1, 2, 3 };
vector<double> v{ 4, 5, 6 };
auto sum_vec = add_them(u, v);        // Error!: + not defined
```

as expected we would get a compiler error. With the Visual Studio compiler, for example, the error message starts off with something that is reasonably helpful, pointing out the `+` operator is not defined in this case, but what follows is a small tsunami of verbosity that does little to help us any further. The following is just an excerpt of the entire compiler output.

```
(Output)
<source>(7): error C2676: binary '+': 'T' does not define this operator or a conversion to a type acceptable to the predefined operator
      with
      [
         T=std::vector<int,std::allocator<int>>
      ]
C:/data/msvc/14.39.33321-Pre/include\xutility(1792): note: could be 'std::reverse_iterator<_BidIt> std::operator +(reverse_iterator<_BidIt>::difference_type,const std::r
<source>(7): note: 'std::reverse_iterator<_BidIt> std::operator +(reverse_iterator<_BidIt>::difference_type,const std::reverse_iterator<_BidIt> &) noexcept(<expr>)': cou
      with
      [
         T=std::vector<int,std::allocator<int>>
      ]

. . .

Compiler returned: 2
```

Using the gcc compiler, the result isn't much better:

```
(Output)
<source>: In instantiation of 'T add_them(T, T) [with T = std::vector<int>]':
<source>:14:28:   required from here
  14 |     auto sum_vec = add_them(u, v);
     |                    ~~~~~~~~^~~~~
<source>:7:19: error: no match for 'operator+' (operand types are 'std::vector<int>' and 'std::vector<int>')
    7 |         return t1 + t2;
      |                ~~~^~~~
In file included from /opt/compiler-explorer/gcc-trunk-20240229/include/c++/14.0.1/bits/stl_algobase.h:67,
                 from /opt/compiler-explorer/gcc-trunk-20240229/include/c++/14.0.1/vector:62,
                 from <source>:1:
/opt/compiler-explorer/gcc-trunk-20240229/include/c++/14.0.1/bits/stl_iterator.h:627:5: note: candidate: 'template<class _Iterator> constexpr std::reverse_iterator<_
  627 |     operator+(typename reverse_iterator<_Iterator>::difference_type __n,
      |     ^~~~~~~~

. . .

Compiler returned: 1
```

In this simple example, it's not difficult to pinpoint the problem from the first few lines in the compiler output, but again in practice, where template programming can become exponentially more complex

very quickly, we often end up having to search for answers obscured in cryptic output that can easily exceed what resulted in this example.

Defining Concepts

C++20 concepts help to minimize the excess, resulting more concise and helpful compiler error information. Continuing with the `add_them(.)` example, we can define a concept, that requires t1 and t2 to be addable, as follows:

```
template<typename T>
concept can_add = requires(const T& t1, const T& t2) { t1 + t2; };
```

A concept definition uses the same template parameter as the function, and assigns the concept name, `can_add` to the requirement (`requires`), which takes the form of a function itself. The concept conditions are then applied to the original function by appending the requirement to the end of the function signature:

```
template<typename T>
T add_them(const T& t1, const T& t2)  requires can_add<T>
{
      return t1 + t2;
}
```

Now, when attempting to apply the `can_add(.)` function to two vectors, the Visual Studio compiler returns the error messages, telling us right up front what the problem is and pointing to the `can_add` concept evaluating to `false` :

```
(Output)

<source>(18): error C2672: 'add_them': no matching overloaded function found
<source>(9): note: could be 'T add_them(T,T)'
<source>(18): note: the associated constraints are not satisfied
<source>(9): note: the concept 'can_add<std::vector<int,std::allocator<int>>>' evaluated to false
<source>(6): note: binary '+': 'const std::vector<int,std::allocator<int>>' does not define this operator or a conversion to a type acceptable to the predefined operator

. . .

Compiler returned: 2
```

With the gcc compiler, we can again get the more relevant information up front, indicating the constraints of the `can_add` concept has not been satisfied.

```
(Output)
<source>:18:28: error: no matching function for call to 'add_them(std::vector<int>&, std::vector<int>&)'
  18 |     auto sum_vec = add_them(u, v);
     |                    ~~~~~~~~^~~~~
<source>:9:3: note: candidate: 'template<class T> T add_them(T, T) requires  can_add<T>'
   9 | T add_them(T t1, T t2) requires can_add<T>
     |   ^~~~~~~
<source>:9:3: note:   template argument deduction/substitution failed:
<source>:9:3: note: constraints not satisfied
<source>: In substitution of 'template<class T> T add_them(T, T) requires  can_add<T> [with T = std::vector<int>]':
<source>:18:28:   required from here
<source>:9:3: note:    18 |     auto sum_vec = add_them(u, v);
<source>:9:3: note:       |                    ~~~~~~~~^~~~~
<source>:6:9:   required for the satisfaction of 'can_add<T>' [with T = std::vector<int, std::allocator<int> >]

. . .

Execution build compiler returned: 1
```

The advantages of concepts, however, are not limited to improving compiler errors. They also make the code itself clearer and more maintainable by specifying argument requirements alongside the function (or class) itself, as should be apparent in the examples that follow.

Defining Concepts with Multiple Conditions

Suppose next we want to have a concept that requires two parameters be both addable and subtractable. This can be done by appending a subtractable condition inside the body of the `requires` block:

```
template<typename T>
concept can_add_and_subtr = requires(const T& t1, const T& t2)
{
      t1 + t2;
      t1 - t2;
};
```

Alternatively, we can have separate concept definitions and then combine them with *and* (`&&`) and *or* (`||`) conditions. For example, we could define a separate concept called `can_subtract` :

```
template<typename T>
concept can_subtract = requires(const T& t1, const T& t2) { t1 - t2; };
```

Then, we could instead use this in an *and* condition with `can_add` to define an equivalent "addable and subtractable" concept:

```
template<typename T>
concept can_add_and_subtr_combined = requires(const T& t)
{
      can_add<T> && can_subtract<T>;
};
```

One more point, the function template parameter can be replaced by the concept itself; eg:

```
template<typename T>
concept can_square = requires(const T& t) { t * t; };

template <can_square T>    // Concept "can_square" replaces the usual function template parameter
T square_it(const T& t) // Can drop the "requires(.)"
{
      return t * t;
}
```

The examples in this book are based on a single template parameter `T`. Concepts can also be extended to two or more template parameters, but the process becomes more complex and can get start getting into the weeds of implicit type conversion and other matters. For readers who wish to pursue this topic, Item 3 in **{8}** is one place to start.

Standard Library Concepts

A set of pre-defined concepts, based on type traits introduced in C++11 **{10}** (Josuttis, SL 2E, Sec 5.4 p 122), is also part of the Standard Library as of C++20. For example, we can enforce the rule that a function will only accept floating point (`double` , `float` , etc) arguments by appending the pre-defined `std::floating_point<T>` concept. To start, consider now a `Quadratic` *class* template that takes in coefficients `a` , `b` , and `c` , and which defines an operator `()` that computes the usual

\begin{equation} ax^2 + bx + c \end{equation}

```
#include <concepts>          // Note we need the Standard Library <concepts> header now

template<typename T> requires std::floating_point<T>
class Quadratic
{
public:
      Quadratic(T a, T b, T c):a_{a}, b_{b}, c_{c}{}
      T operator ()(T x) const
      {
            return (a_ * x + b_) * x + c_;
      }
```

```
private:
    T a_, b_, c_;
};
```

This would now ensure that if we tried to create a `Quadratic` with `T` = `int`,

```
Quadratic<int> quad_int(2, 4, 2);
```

the compiler would complain, but in a good way such that it would be more obvious what the problem is.

If we wanted the class to also accept integer types (`int`, `unsigned`, etc), this is easily rectifiable by appending an *or* condition with the integer analog of `std::floating_point<T>`, namely `std::integral<T>`.
This is done again by first defining a concept, such as the following:

```
template<typename T>
concept Number = requires(T t)
{
    std::integral<T> || std::floating_point<T>;
};
```

Then, in place of the `requires` statement at the top of the class definition, we can just replace its single template parameter with the `Number` concept:

```
template<Number N>
class Quadratic
{
public:
    Quadratic(N a, N b, N c) :a_{ a }, b_{ b }, c_{ c } {}

    N operator ()(N x) const
    {
        return (a_ * x + b_) * x + c_;
    }

private:
    N a_, b_, c_;
};
```

Now, if we try to use invalid arguments at construction, for example `string` types:

```
Quadratic<string> quad_string{ "this", "won't", "compile" };
```

compilation will fail (as desired), but with a descriptive yet concise error message that tells us what the problem is.

One last point to note is the `Number` concept above could alternatively be defined in a more compact form if desired:

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

Summary

Modules were a long awaited enhancement that are now part of C++20.

There are four different types of module files:

- Primary module interface unit
- Module implementation unit
- Module interface partition unit
- Internal module partition unit

We mainly covered the first two unit types in this list, as partitions require a more advanced and detailed discussion. Still, there is a lot that can be accomplished with primary module interface and module implementation units. Module interfaces can be used in place of traditional header files, and thus pesky leakages caused by `#include`, and additional compile time due to redundant parsing of header code can be avoided. Resulting code is also cleaner in the sense that with modules there is no need for "ugly" preprocessor statements.

These interfaces also give the programmer more control over which functions and classes are to be accessible outside of the module by explicit designation using the `export` keyword. Declarations and implementations can also be placed in separate interface and implemenation files similar to existing common practice while still taking advantage of the safeguards and organization provided by modules.

Concepts were also a well-anticipated addition to C++20 and are fully implemented in all three major C++ distributions. They are being graciously welcomed by programmers who rely heavily on template programming, as they pinpoint errors more precisely at compile time while preventing the passage of invalid template parameter types. Concepts can be user-defined, while there is also a set of pre-defined concepts provided in the Standard Library. Both categories can be combined with logical *and* and *or* conditions.

References

{1} Josuttis, C++20, Ch 16

{2} Gabriel Dos Reis & Cameron DaCamara, *Implementing C++ Modules: Lessons Learned, Lessons Abandoned*, CppCon 2021:

https://cppcon2021.sched.com/event/nv1r/implementing-c-modules-lessons-learned-lessons-abandoned

Slides: *https://cppcon.digital-medium.co.uk/wp-content/uploads/2021/10/CppCon2021-Implementing-C-Modules.pdf*

Video: *https://www.youtube.com/watch?v=BFXSaUMi4vY*

{3} ISO C++ Proposal P2412: A Standard-Module Version of Standard Library Headers *https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2412r0.pdf* (*https://wg21.link/p2412*)

{4} ISO C++ Proposal P2465: Standard-Module Version of Standard Library Headers planned for 2023 *https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2465r3.pdf* (*https://wg21.link/p2465*)

{5} ISO C++ Proposal P1502: Standard Library header units *https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1502r1.html*

{6} (Grimm C++20), "ugly" include guards

{7} Stack Overflow: One Definition Rule (ODR) violations *https://en.cppreference.com/w/cpp/language/definition*

{8} Gajendra Gulgulia, C++20 Concepts, Item 3

{9} op cit {3} (Josuttis C++20), Secs 16.2.3 -16.2.4, pp 571-75

{10} Josuttis, *The C++ Standard Library (2E)*, Sec 5.4 p 122