

KIV/CPP – Programming in C++

09. Concepts, coroutines, modules

Martin Úbl

KIV ZČU

2025/2026

- extension of the type and template system with the ability to name requirements on a type (since C++20)
- concept definition
 - the `concept` keyword
 - always used together with a template declaration
 - existing concepts can be used as a basis
 - more advanced requirements can be defined
 - e.g., “has a method `xyz()` that returns `double`”
 - they can be composed as logical expressions (`&&`, `||`)
- applying a concept
 - before a type identifier (template type or `auto`)
 - in the definition of template parameters

- example – a simple concept, a requirement that the type is a signed integral value
- definition:

```
template<typename T>
concept SignedInteger = std::is_integral_v<T>
                        && std::is_signed_v<T>;
```

- usage 1: “*abbreviated function template*” (probably the clearest):

```
SignedInteger auto Neg(SignedInteger auto in) {
    return -in;
}
```

- usage 2: “constrained template parameter”:

```
template<SignedInteger T>
T Neg(T in) {
    return -in;
}
```

- so far these forms have not had the shape of a so-called *constraint*, although they are constraints in practice
- *constraints* are a principle closely related to *concepts*
 - one can say that they are the practical application of *concepts*

- the `requires` keyword – defines a so-called *constraint*
- usage 3 (`requires` – constraint on a template parameter):

```
template<typename T>
    requires SignedInteger<T>
T Neg(T in) {
    return -in;
}
```

- usage 4 (`requires` – constraint on a function):

```
template<typename T>
T Neg(T in) requires SignedInteger<T> {
    return -in;
}
```

Concepts

- for comparison – the same construction for SFINAE without concepts:

```
template<class T,  
        class = typename std::enable_if<  
                        std::is_integral_v<T>  
                        && std::is_signed_v<T>>  
        ::type >  
T Negate(T in)  
{  
    return -in;  
}
```

- difficult-to-read notation – understanding it requires knowing SFINAE
- there are several completely different ways to implement SFINAE
- in contrast, concepts are a much cleaner solution

- concept definition
- (1) by composing already existing concepts or constexpr definitions convertible to bool

```
template<typename T>  
concept SignedInteger = std::is_integral_v<T>  
                        && std::is_signed_v<T>;
```

```
template<typename T>  
concept SignedLargeInteger = SignedInteger<T>  
                             && sizeof(T) > 4;
```

- concept definition
- (2) by creating a set of requirements, the `requires` keyword

```
template<typename T>
concept HasToString = requires(T t) {
    { t.to_string() }
    -> std::convertible_to<std::string>;
};
```

- “type `T` has a method `to_string()` whose return value is convertible to `std::string`”
 - a “pseudo-instance” `t` is used to evaluate the syntax

Concepts

- concept definition
- (2) by creating a set of requirements, the `requires` keyword

```
template<typename T>
concept Addable = requires(T t) {
    t + t;
};
```

- “type `T` has an operator defined for addition with itself”

```
template<typename T>
concept AddableWithSelf = requires(T t) {
    { t + t } -> std::convertible_to<T>;
};
```

- “type `T` has an operator defined for addition with itself and the result is a type convertible to `T`”

- concept definition
- it can itself be template-parameterized

```
template<typename T, typename U>
concept MultipliableWith = requires(T t, U u) {
    t * u;
};
```

- “type T has an operator defined for multiplication with type U”

```
double InchToMeters(MultipliableWith<double>
                    auto in) {
    return in * 0.0254;
}
```

- a concept defined by a *constraint* can also be understood as “if I write this, it compiles”

```
template<typename T>  
concept C = requires(const char* u) {  
    T{u};  
};
```

- “if I have type T and want to construct it like this from const char*, it will work”

Concepts

- multiple requirements (*constraints*) in one concept
- separated by semicolons

```
template<typename T>
concept AddableMultipliable =
    std::convertible_to<T, double>
    && requires(T t) {

        { t + t } -> std::convertible_to<T>;
        { t * t } -> std::convertible_to<T>;
    };

```

- “type T is convertible to double and has addition and multiplication operators whose result is convertible to T”

- why do we need to say “is convertible”?
- we need a complete semantic expression – after the arrow there is something that must evaluate to true or false at compile time

```
{ t + t } -> T    // does not work
```

```
{ t + t } -> std::same_as<T>;
```

```
{ t + t } -> std::convertible_to<T>;
```

- chaining concepts and function overloading
- let us have concepts A and B
 - concept B extends concept A
 - concept B does not add conflicting requirements
 - then concept B is so-called *stronger* than concept A
- we can then define overloaded functions for concepts A and B at the same time
- the compiler then selects the strongest available concept

Concepts

```
template<typename T>
concept SignedInteger = std::is_integral_v<T> && std::is_signed_v<T>;

template<typename T>
concept SignedLargeInteger = SignedInteger<T> && sizeof(T) > 4;

void Test1(SignedInteger auto a) {
    std::cout << "SignedInteger:␣" << a << std::endl;
}

void Test1(SignedLargeInteger auto a) {
    std::cout << "SignedLargeInteger:␣" << a << std::endl;
}

// ...

Test1(55);
Test1(99LL);

// SignedInteger: 55
// SignedLargeInteger: 99
```

- in some cases, a concept can substitute for an *interface* element
 - e.g., if we had an interface only because of one or two concrete implementations
- in C++, that would otherwise be substituted by abstract classes
- abstract classes are, however, an element of dynamic polymorphism
- concepts (and templates) are purely a compile-time matter
- somewhat more understandable than static polymorphism/CRTP
- they can be combined generically with variadic templates

- coroutines (since C++20)
- partly supported in the language, partly in the standard library
- coroutines are functions that can be suspended at an agreed point
- they generate results incrementally
- new keywords – syntactically they are operators
 - `co_yield` – suspends the coroutine and returns a result
 - `co_await` – passes control to another coroutine with the intention of continuing execution of the current one in the future
 - `co_return` – terminates the coroutine and returns a result
- unfortunately, implementing the logic is purely a “do it yourself” matter
 - hopefully, the C++2a standard will also bring library support

Coroutines

- coroutines
- “the nicer part” I. – value generator

```
generator<int> counter()
{
    for (int i = 0; i < 3; i++) {
        std::cout << "Yielding:_" << i << std::endl;
        co_yield i;
    }
}

auto gen = counter();
while (gen)
    std::cout << "Receiving:_" << gen() << std::endl;

// Yielding: 0
// Received: 0
// Yielding: 1
// Received: 1
// Yielding: 2
// Received: 2
```

- coroutines
- “the nicer part” II. – deferred computation/task

```
lazy<int> lazy_task() {  
  
    int result;  
  
    // ...  
  
    co_return result;  
}
```

- coroutines
- “the nicer part” III. – a suspendable function of some processing loop

```
task<> lazy_task() {  
    Buffer buf;  
    while (running) {  
        // read from input  
        size_t count = co_await ReadFile(file, buf);  
        // process  
        Process(buf);  
        // write to output  
        co_await WriteFile(file, buf);  
    }  
}
```

- coroutines
- their strength lies in the fact that, de facto, this is still one and the same function call
- e.g., the processing loop mentioned above
 - `lazy_task`, `ReadFile`, and `WriteFile` are also coroutines
 - it can be understood as all functions being called exactly once
 - only the execution state is “advanced” each time
- coroutines never run in parallel
 - they pass control to each other completely deliberately (cooperative multitasking)
 - execution ends when:
 - the corresponding `coroutine_handle` expires
 - the coroutine calls `co_return`
 - an exception is thrown

- coroutines
- unfortunately, the generator, lazy, and task types must be supplied
- the standard defines requirements on methods, member attributes, and types
 - `promise_type` – synchronization promise type
 - defines functions for the coroutine return value (generator)
 - passing exceptions that occur during generation
 - passing the return value (the value itself)
 - and others...
- typically, we then want some interface for accessing elements – we define that ourselves
- some compilers and standard library implementations already contain some experimental implementations
 - clang/stdc++ – `std::experimental::task`
 - MSVS – `std::experimental::generator`
 - most likely in some form in C++2a or later

- `promise_type` – definition requirements
 - `get_return_object()` – defines the coroutine return value
 - `initial_suspend()` – defines the suspend policy when the coroutine starts
 - `final_suspend()` – defines the suspend policy when the coroutine terminates
 - `yield_value(Type value)` – called in `co_yield` to pass a value
 - `unhandled_exception()` – called when an unhandled exception occurs
 - `return_void()` – for calling `co_return` without a value
 - `return_value(Type value)` – for calling `co_return` with a value
 - if the coroutine contains `co_return`, exactly one of the `return_void` and `return_value` methods must be defined

- a class on which `co_await` can be called
 - `await_ready` – defines the suspend policy at the moment `co_await` is called
 - `await_suspend` – called at the moment the coroutine is suspended when calling `co_await`
 - `await_resume` – called when execution resumes (the called coroutine returned a value); the return value is the result of the `co_await` operator

- generator example, part I.

```
template<typename T>
struct generator {
    struct promise_type;
    using handle_type = std::coroutine_handle<promise_type>;

    struct promise_type {
        T value_;
        std::exception_ptr exception_;

        generator get_return_object() {
            return generator(handle_type::from_promise(*this));
        }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void unhandled_exception() { exception_ = std::current_exception(); }

        template<std::convertible_to<T> From>
        std::suspend_always yield_value(From&& from) {
            value_ = std::forward<From>(from);
            return {};
        }
        void return_void() {}
    };

    handle_type h_;

    generator(handle_type h) : h_(h) {}
    ~generator() { h_.destroy(); }
```

- generator example, part II.

```
explicit operator bool() {
    fill();
    return !h_.done();
}

T operator()() {
    fill();
    full_ = false;
    return std::move(h_.promise().value_);
}

private:
    bool full_ = false;

    void fill() {
        if (!full_) {
            h_();
            if (h_.promise().exception_)
                std::rethrow_exception(h_.promise().exception_);
            full_ = true;
        }
    }
};
```

- exhaustive list of requirements incl. semantics:
 - `https://en.cppreference.com/w/cpp/language/coroutines`
- in this course, we will not discuss coroutines further for now
- among other reasons, because at the time this presentation was prepared no compiler had complete support
- and also because the standard library does not have support structures for convenient implementation

- why coroutines, and why not `std::async` (`std::future`)?
- `std::async` adds a large amount of overhead when called repeatedly
 - coroutines keep exactly one context, which is only suspended
 - `std::async` creates and destroys a context on every call
- coroutines are more efficient
- however, at least for now, they are not clearer

- making work more convenient and substituting for what the standard library does not provide (yet)
- <https://github.com/lewissbaker/cppcoro>
 - implementations of generators, tasks, and others
 - awaitable mutexes, semaphores, ...
 - implementations of various asynchronous I/O operations in coroutine style (so that `co_await` can be used on them)

- *Modules* (since C++20)
- independently compilable units
- logical separation of components
- in a certain way, they consistently combine into one mechanism what could otherwise be defined anywhere and implemented anywhere as well
 - a (forward) declaration can actually be anywhere; a header file is only a very common place to put it
 - an implementation likewise only needs to be “somewhere” (where the linker can find it)
- modules have an interface and an implementation
 - interface and implementation is tied together

- *Modules* (since C++20)
- modules are compiled and included exactly once
- compared to the original method with a preprocessor macro
 - `#include <header.h>`
 - inside, either `#pragma once` or a macro preventing multiple inclusion
- they partially replace the *precompiled headers* mechanism

- *Modules* (since C++20)
- module interfaces
 - replace what were conventionally header files
 - contain the `export module Name` clause
 - have their own file “type”
 - `.ixx` extension in MSVS
 - `.cppm` extension in clang and GCC
- module implementations
 - replace what was conventionally a source file named the same as the header
 - contain the `module Name` clause
 - it is already classically in a `.cpp` file

- basic module example
- interface, `calc.ixx`

```
export module calc;
```

```
export double pow2(double in);
```

- implementation, `calc.cpp`

```
module calc;
```

```
double pow2(double in) {  
    return in * in;  
}
```

- what can be exported?
 - functions
 - classes
 - namespaces
 - type aliases
 - variables
 - etc...
- what cannot be exported?
 - anonymous namespaces
 - static variables
 - anything that has been declared in advance as non-exported

Modules

- module example – exporting a class
- interface, calc.ixx

```
export module calc;
```

```
export class calculator {  
    public:  
        static double pow2(double in);  
};
```

- implementation, calc.cpp

```
module calc;
```

```
double calculator::pow2(double in) {  
    return in * in;  
}
```

- module partitioning
- modules can essentially be divided hierarchically into parts
- the import structure can be compared to inclusion
- only, of course, it benefits from the advantages of modules
- a *partition* is introduced by a colon
 - if the main module is named `test`, then its partitions can be `test:first`, `test:second`, ...
- do not confuse this with a dot – that can be used for name separation, not for partitioning

- example
- module interface, `calc_basic.ixx`

```
export module calc:basic;
```

```
export double Add(double a, double b);
```

```
export double Sub(double a, double b);
```

- module interface, `calc_powers.ixx`

```
export module calc:powers;
```

```
export double Pow2(double a);
```

```
export double Pow3(double a);
```

- example
- module interface, `calc.ixx`

```
export module calc;
```

```
export import :basic;
```

```
export import :powers;
```

- the implementation of modules `calc_basic.cpp` and `calc_powers.cpp` can be easily inferred
- the syntax `export import Name` means that we import the module (or partition) with the given name and also export its exported interface from the current module

- implicit export
- if a namespace is exported, its contents are implicitly exported
- analogously, if one of the namespace contents is exported, the namespace is implicitly exported as well

```
export namespace ns {  
    void something() { /* ... */ }  
}
```

```
namespace ns {  
    export void something() { /* ... */ }  
}
```

- importing
- in modules, `import` can appear before any module declaration
- `import` can only be at global scope (it cannot be imported later in a function body, etc.)
- only partitions belonging to the given module can be imported
- `import` cannot import the same module in which it appears

Modules

- visibility vs. reachability
- visible identifier = it can be found by its name
- reachable identifier = what it denotes can be used
- a module exports a function that returns an instance of a class, but does not export the class
 - the function is visible
 - the class is only reachable

```
class PrivateClass {  
    // ...  
};
```

```
export PrivateClass create_priv() {  
    return PrivateClass{ };  
}
```

- the function can be called and the returned instance can be manipulated as if the type were visible
- however, the class cannot be instantiated by its identifier
- with a trick, however, additional instances of this class can be created

```
// ok, 'p' will be of type PrivateClass  
auto p = create_priv();
```

```
// wrong! We cannot see PrivateClass  
PrivateClass pp{};
```

```
using SecretClass = decltype(p);  
// ok, 'q' has the same type as 'p'  
SecretClass q{};
```

- global module
- once we start using modules in C++, everything must be part of some module
- a global module is implicitly defined
 - it has no name
 - it cannot be imported
 - everything declared outside a module implicitly belongs to it
 - the `main()` function may only be here

- header-unit import
- modules assume that we will want to “modularize” something that is not a module
- e.g., an already existing pair of a header file and the corresponding source file
- then it is enough to import the header file

```
export module security;
```

```
import <openssl/ssl.h>;
```

- the header file is included and all its internal definitions are exported

- header-unit import
- note – imports are not affected by defined macros
- therefore a header file cannot be “parameterized” by the preprocessor state

```
export module winmodule;
```

```
#define _UNICODE  
import <windows.h>;
```

- modules (module units) are linked differently and elsewhere, so the `_UNICODE` macro will have no effect here

- from C++23/26 onward, there is a plan to modularize the standard library
- then header-unit import will not be needed for it
- compilation should become faster
- MSVS already provides part of the standard library as modules:
 - it must be explicitly installed
 - `import std.core;`
 - `std.threading, std.regex, std.filesystem, ...`
 - <https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp>

- modules are part of the effort to eliminate the need for the preprocessor
- among other things, this effort includes:
 - `constexpr`, `enum class` – replacement for `#define` for constants
 - `if constexpr` – replacement for `#if` and `#ifdef`
 - templates, concepts – replacement for textual code substitution
- the preprocessor has its “own language” above C/C++ itself
- it potentially complicates translation, code orientation, scalability, and safety
 - e.g., MSVS and the implicitly defined `min (max)` macro
 - the compiler emits a highly nonspecific error if you use `std::min` somewhere in the code
 - `min` is replaced by the macro contents, effectively inserting the ternary operator
 - compilation fails because `std::(x>y?y:x)` cannot be compiled

- general problems of the preprocessor:
 - it has no semantics
 - it does not know data types
 - spaces and whitespace can break code
 - it does not participate in compilation and cannot “communicate” with the code it generates
- however, it is still needed
 - before complete support for modules
 - for versioning selected multitargeting libraries
 - e.g., `#define CL_TARGET_OPENCL_VERSION 120`
 - compatibility with C source code when sharing code
 - a certain part of the debugging process
 - `__FILE__`, `__LINE__`, ...