

KIV/CPP – Programming in C++

10a. Additional C++ Language Constructs

Martin Úbl

KIV ZČU

2025/2026

- overview of constructs not covered so far
- additional STL specialties
- features from C++14, C++17, C++20, and C++23

Additional C++ Language Constructs

- *small string optimizations*
- `std::string` optimization for small strings
- normally, dynamic allocation would be needed
- since C++17, `std::string` allows storing short strings directly inside itself
 - 15 characters for MSVS
 - 23 characters for GCC/clang
- some similarity to a fast symbolic link in VFS/ext

```
// without dynamic allocation
```

```
std::string a{"Hello"};
```

```
// with dynamic allocation
```

```
std::string b{"Could_somebody_tell_me_what_the_f
```

Additional C++ Language Constructs

- *string views*
- `#include <string_view>`
- the `std::string_view` class provides a “view” of other string memory
- it does not store the string itself
- the difference can be seen, for example, when calling `substr`
 - `std::string` returns `std::string` - $O(n)$
 - it returns a “copy” of the substring
 - `std::string_view` returns `std::string_view` - $O(1)$
 - it returns only a view of the substring (essentially two pointers)
- useful, for example, when parsing inputs

```
std::string s{ "string" };  
std::string_view sv(s.c_str(), 3);  
std::cout << sv << std::endl;
```

```
// 2 instance std::string
// string copied twice
std::string s{ "some_very_long_string_that_does_"
auto s2 = s.substr(7);
```

```
// 2 instance std::string_view
// but only one string instance
std::string_view sv(s);
auto sv2 = sv.substr(7);
```

- `#include <optional>`
- the `std::optional` class wraps another type whose instance may or may not be provided
- template type
- constructor with an instance of the given type or a default constructor
- `std::nullopt` can be used to create an empty value

```
std::optional<int> GetValue(size_t idx) {  
    if (myMap.find(idx) != myMap.end())  
        return myMap[idx];  
    return std::nullopt;  
}
```

- `std::optional`
- checking whether it has a value
 - `has_value()` method
 - overloaded `bool()` operator
- retrieving the value
 - `value()` method
 - overloaded dereference operator

Additional C++ Language Constructs

- `#include <variant>`
- the `std::variant` class wraps one value of several possible types
- type-safe C-style union
- template type (variadic), with all types it can potentially store listed in the template
- it always stores exactly one of them (assignment, constructor)
- retrieval using the `std::get<T>` function
- getting the wrong type throws a `std::bad_variant_access` exception

```
std::variant<int, double, long long> var;
```

```
var = 1;
```

```
var = 15.4;
```

```
var = 99999LL;
```

- `std::variant`
- visitor pattern

```
std::variant<int, double, long long> var = ...;
```

```
std::visit([&](auto&& value) {  
    // "value" has the correct type  
}, var);
```

- for the given variant, the correct value type is deduced, and the corresponding lambda is therefore specialized (auto parameter)
- useful, for example, for traversing a vector of variants
- note: `std::variant` uses only the stack (never the heap)

Additional C++ Language Constructs

- `#include <any>`
- the `std::any` class wraps one value of any type
- type-safe C-style `void*`
- it is not a template type
- allocation is always on the heap
- retrieval by casting with `std::any_cast<T>`
- casting to the wrong type throws a `std::bad_any_cast` exception
- the type must be copyable

```
std::any v;
```

```
v = 42;
```

```
v = "string";
```

- `std::any`

```
std::any v = ...;
```

```
try {  
    int ival = std::any_cast<int>(v);  
}  
catch (std::bad_any_cast& bac) {  
    // ...  
}
```

- slower than `std::variant`
 - dynamic deduction at run time
 - heap allocation

- binary literals

```
uint8_t bin = 0b00010011;
```

- digit separators

```
uint32_t i1 = 1'005'999;  
uint16_t i2 = 0b00001111'10110110;
```

- `std::string` construction
- `using namespace std::string_literals;`

```
auto str = "hello"s;
```

Additional C++ Language Constructs

- user-defined literals
- possibility to define custom literals
- e.g. unit conversions
- they “look” like the "" operator
- user-defined literals should start with an underscore
 - names without an underscore are reserved for the language standard

```
// e.g. everything to meters
constexpr long double operator"" _cm(
    long double cm) {
    return cm / 100.0;
}

long double A4_height = 29.7_cm; // 0.297
```

- can be defined only for selected types
 - `const char*`
 - `long double`
 - `unsigned long long int`
 - `char`, `wchar_t`, `char16_t`, ...
- however, they may return any type, even a class instance

- `#include <chrono>`
- standardization of time units and intervals
- working with time values
- `std::chrono::duration<T>` – a time interval with the selected type
 - float, double, ...
 - independent of units (seconds, minutes, ...)
- `std::chrono::time_point<T>` – value of the selected clock

- `steady_clock`
 - non-decreasing clock, does not represent real time
 - constant time between ticks
 - best suited for measuring time intervals, e.g. computation time
- `system_clock`
 - represents real time
- `high_resolution_clock`
 - high-resolution clock, does not necessarily represent real time
- `now()` method for obtaining the value
- support for addition, subtraction, ...
- conversion to required units –
`std::chrono::duration_cast<T>`

- `#include <chrono>`
- `using namespace std::chrono_literals`
- defined literals for time intervals
 - ns, us, ms, s, min, h

```
auto duration = 150ms;
```

```
std::this_thread::sleep_for(250ms);
```

- `std::filesystem`
- `#include <filesystem>`
- since C++17
- working with the file system using a unified interface
 - working with files, directories, and links
 - permissions
 - modification and creation times, ...
 - partition usage and capacity

Additional C++ Language Constructs

- `std::filesystem::path`
- represents a path in the file system
- does not necessarily identify an existing file/folder/link
- absolute/relative
- conversion to absolute path with `std::filesystem::absolute`
- conversion to relative path with `std::filesystem::relative`
- can be implicitly converted to `std::string`

```
std::filesystem::path filepath("input.txt");
```

```
std::cout << filepath << std::endl;  
std::cout << std::filesystem::absolute(filepath)  
          << std::endl;
```

Additional C++ Language Constructs

- `std::filesystem::path`
- appending a component using `append` or the `/` and `/=` operators

```
std::filesystem::path filepath("C:\\");
```

```
filepath.append("folder");
```

```
std::cout << filepath << std::endl;  
// C:\folder
```

```
filepath /= "subfolder";
```

```
std::cout << filepath << std::endl;  
// C:\folder\subfolder
```

- `std::filesystem::path`
- additional methods
 - `filename` – returns the file name
 - `parent_path` – removes the last component (returns the parent directory)
 - `make_preferred` – converts path component separators to the preferred ones for the given OS
 - `remove_filename` – removes the file name from the path
 - `replace_filename` – replaces the file name with another name
 - `stem` – returns the file name without the extension
 - `extension` – returns the extension

- `std::filesystem`
- additional functions – system paths
 - `std::filesystem::current_path()` – returns or sets the working directory
 - `std::filesystem::temp_directory_path()` – returns the directory for temporary files

- `std::filesystem`
- additional functions - file system manipulation
 - `std::filesystem::exists()` – existence of a file/folder/link
 - `std::filesystem::copy()` – copy of a folder or file
 - `std::filesystem::rename()` – rename or move
 - `std::filesystem::remove()` – deletion of a file or folder
 - `std::filesystem::status()` – returns file properties

- `std::filesystem`
- `std::filesystem::directory_entry` – class representing an entry in a directory
 - the `path()` method returns the physical path
- `std::filesystem::directory_iterator` – returns a class instance for iterating over directory entries

- `std::filesystem`
- some functions throw a `std::filesystem::filesystem_error` exception on failure
 - access to a file that does not exist
 - system files
 - permissions

Additional C++ Language Constructs

- `if constexpr`
- condition evaluable at compile time
- a “less verbose” variant of, for example, template instantiation
- the condition must be `constexpr`
- this can avoid SFINAE and `std::enable_if`

```
template<typename T>
void DoSomething(const T& val) {

    if constexpr (std::is_integral_v<T>)
        DoIntegral(val);
    else
        DoOther(val);

}
```

- attributes
- syntactic addition for further optimizations
- marking a function/method/code block with certain properties
 - `[[noreturn]]` – the function will not return (the only return is either by exception or not at all)
 - `[[deprecated("reason")]]` – the function is marked as deprecated
 - `[[fallthrough]]` – a switch case level is intentionally not ended with break
 - `[[maybe_unused]]` – marks an identifier that may not be used anywhere
 - `[[likely]]`, `[[unlikely]]` – e.g. for conditions - when we know that a condition will be very often true, we mark the block below it as likely (optimization)

- marking a function/method/code block with certain properties
 - `[[assume(...)]]` – assumption that the expression evaluates to true (optimization)
 - others may be compiler-specific, e.g.
`[[gnu::always_inline]]`
 - and more...
 - en.cppreference.com/w/cpp/language/attributes

- *structured binding*
- multiple initialization can be bound to an object
- e.g. copy an array into multiple variables
- or replace `std::tie` for `std::tuple`
- syntax with `auto[...] = ...`
- can bind by value or by reference

```
std::array<int, 2> arr{ 5, 10 };  
auto[a, b] = arr;
```

```
std::tuple<int, double> tup{ 5, 12.5 };  
auto&[i, d] = tup;
```

- *structured binding*
- possible, for example, to check whether inserting an element into a container succeeded

```
std::map<int, std::string> mp;  
  
if (auto [itr, succ]  
    = mp.insert({ 42, "meaning" }); succ)  
    std::cout << "OK" << std::endl;  
else  
    std::cout << "FAIL" << std::endl;
```

- *structured binding*
- can bind by value, l-value reference, or r-value reference

```
std::array<int, 2> arr{ 5, 10 };  
auto[a, b] = arr;  
auto&[c, d] = arr;  
auto&&[e, f] = std::make_tuple(5, 6);
```

Additional C++ Language Constructs

- `std::numeric_limits`
- `#include <limits>`
- standard template variant of numeric constants for different types
 - maximum and minimum representable value
 - infinity and Not-a-Number values
 - various HW-related properties

```
double maximum =  
    std::numeric_limits<double>::max();  
  
bool exact =  
    std::numeric_limits<double>::is_exact();  
// false :(
```

- Regular expressions
- `#include <regex>`
- classic form of regular expressions
- a set of classes and algorithms

- Regular expressions
- `std::regex`
- represents exactly one regular expression
- the constructor accepts the expression text and flags
- flags (one or more), can be combined with the `|` operator
 - `std::regex::icase` – ignores letter case
 - `std::regex::ECMAScript` – uses regex syntax from the ECMAScript specification
 - `std::regex::awk` – uses awk regex syntax
 - `std::regex::multiline` – matching across multiple lines
 - etc...
 - https://en.cppreference.com/w/cpp/regex/basic_regex

- `std::regex_match` – match of the entire input

```
std::string valid_email{ "fanda@domena.cz" };  
std::string invalid_email{ "aa@bb@domena.cz" };
```

```
const std::regex r(  
    "[a-z0-9\\.]+@[a-z0-9\\.]+\\. [a-z]+",  
    std::regex::icase | std::regex::ECMAScript  
);
```

```
std::regex_match(valid_email, r);    // true  
std::regex_match(invalid_email, r); // false
```

- note: regex for e-mails is naturally more complicated

Additional C++ Language Constructs

- `std::regex_match` – match of the entire input + sub-matches
- `std::smatch`

```
const std::string dom{ "www.zcu.cz" };  
const std::regex r(  
    "[a-z+)]\\.([a-z+)]\\.([a-z+)]"  
);
```

```
std::smatch sm;  
if (std::regex_match(dom, sm, r)) {  
    for (auto& s : sm)  
        std::cout << s.str() << std::endl;  
}
```

- the submatch object always contains the whole matched string at the first position, followed by sub-matches

- `std::regex_search` – match part of the input

```
const std::string info{
    "Name:␣Gordon,␣Surname:␣Freeman,␣Age:␣44"
};

const std::regex r("(Surname:␣)([a-z]+)",
    std::regex::icase | std::regex::ECMAScript
);

std::smatch sm;
if (std::regex_search(info, sm, r)
    && sm.size() == 3) {
    // sm[2].str() == "Freeman"
}
```

- `std::regex_replace` – match and replace parts of the input

```
const std::string in{ "miniature_porcupine" };
const std::regex r("i",
    std::regex::icase | std::regex::ECMAScript);

auto res = std::regex_replace(in, r, "y");
// "mynyature porcupyne"

auto res2 = std::regex_replace(in, r, "_$&_");
// "m_i_n_i_ature porcup_i_ne"
```

- Regular expressions
- can be combined and chained in various ways
- regex support in C++ should provide what we are used to from other languages

Additional C++ Language Constructs

- Value array
- `#include <valarray>`
- container for numbers supporting bulk mathematical operations over elements
- internally contiguous memory
 - behaves like a restricted `std::vector`
- mathematical functions added for this type
 - e.g. `std::pow`, `std::sqrt`, and others
- supports slicing (cutting in an interval with a specified stride)
- gives a large space for optimizations
 - cache usage
 - vector instructions (SIMD)
- limited assortment of operations
 - many more options, for example, in the Eigen3 library

- e.g. solving 6 quadratic equations at once

```
// ax^2 + bx + c = 0
std::valarray<double> a{ 1, 1, 2, 2, 3, 3 };
std::valarray<double> b{ 1, 2, 1, 2, 1, 2 };
std::valarray<double> c{ 0, 1, 0, 1, 0, 1 };

// ( -b +/- sqrt(b*b - 4ac) ) / 2a
auto x1 = (-b + std::sqrt(b * b - 4 * a * c))
          / 2 * a;
auto x2 = (-b - std::sqrt(b * b - 4 * a * c))
          / 2 * a;
```

- *complex*
- `#include <complex>`
- support library for working with complex numbers
- effectively just a container over a pair of double numbers
- `std::complex_literals` namespace
 - defines the literal for the imaginary unit `i`
- template type `std::complex`
- classic notation, e.g. `2.0 + 3.0i`
- extension of standard mathematical functions
- we will not analyze it further; it is simply good to know that C++ can do this

- example

```
using namespace std::complex_literals;
```

```
auto c1 = 2.0 + 1.0i;
```

```
auto c2 = 1.0 - 2.0i;
```

```
auto res = c1 * std::pow(c2, 2);
```

```
std::cout << res.real() << "␣+␣"
```

```
      << res.imag() << "i"
```

```
      << std::endl;
```

```
// -38 + 41i
```

- *feature testing* (since C++20)
- imposes an obligation on the compiler to declare which parts of the language and library standard it already supports
- this can improve backward compatibility
- the compiler defines macros including the support “version” (if the standard was revised)
- this can then be checked in code with a classic `#ifdef`
- list of macros
 - https://en.cppreference.com/w/cpp/feature_test

- what about constexpr in VS2019?

```
std::cout << __cpp_constexpr << std::endl;  
// 201907
```

__cpp_constexpr	constexpr	200704L	(C++11)
	Relaxed constexpr, non-const constexpr methods	201304L	(C++14)
	Constexpr lambda	201603L	(C++17)
	Trivial default initialization and asm-declaration in constexpr functions	201907L	(C++20)

- `std::span`
- wraps a range (static or dynamic)
- a “rawer” variant of views – fewer restrictions
- can point to a range of elements in a container, but also in a plain array

```
int* arr = new int[4];
```

```
// statically
```

```
std::span<int, 4> sp1 = arr;
```

```
// dynamically
```

```
std::span<int> sp1 = arr;
```

- `#include <bit>`
- header library for working with data at the bit level
 - `bit_cast` – cleaner variant of dereferencing a reinterpret cast result
 - `popcount` – returns the number of one bits
 - `rotl`, `rotr` – left or right bit rotation
 - `countl_...` – returns the number of leading/trailing ones/zeros
 - and more...
- <https://en.cppreference.com/w/cpp/header/bit>

- `#include <numbers>`
- numeric constants
 - `std::numbers::pi` – Ludolph's number π
 - `std::numbers::e` – Euler's constant e
 - `std::numbers::sqrt2` – square root of two $\sqrt{2}$
 - and more...
- <https://en.cppreference.com/w/cpp/header/bit>

- other, uncategorized
 - `std::midpoint` – computation of the midpoint without overflow risk
 - `std::to_array` – conversion of a container to an array
 - `contains()` on a map and set – replacement for `find(x) == end()`
 - `starts_with()`, `ends_with()` on a string
 - `std::byte` – type representing a byte (not a character type)
 - and many more...

- C++23
 - `std::print`, `std::println` – like `std::format`, but writes directly to a stream
 - `std::flat_map`, `std::flat_set`, ... – variant of classic containers, but separates references to keys and values so that it performs additional indirection internally (removes the need to move the key and value)
 - faster insert, delete, and iteration
 - smaller memory footprint
 - `std::expected` – for better control flow when errors occur
 - `std::mdspan` – multidimensional span
 - it is therefore possible, for example, to have a `std::vector` (1D) of size $M \times N$
 - and use `std::mdspan` to iterate as over a 2D array using two indices

- C++23
 - modularized standard library
 - introduced into the standard
 - compilers and libraries are still lagging behind
 - "deducing this" – reducing code bloat when distinguishing const/non-const and l- and r-value referenced contexts is needed
 - and many more...