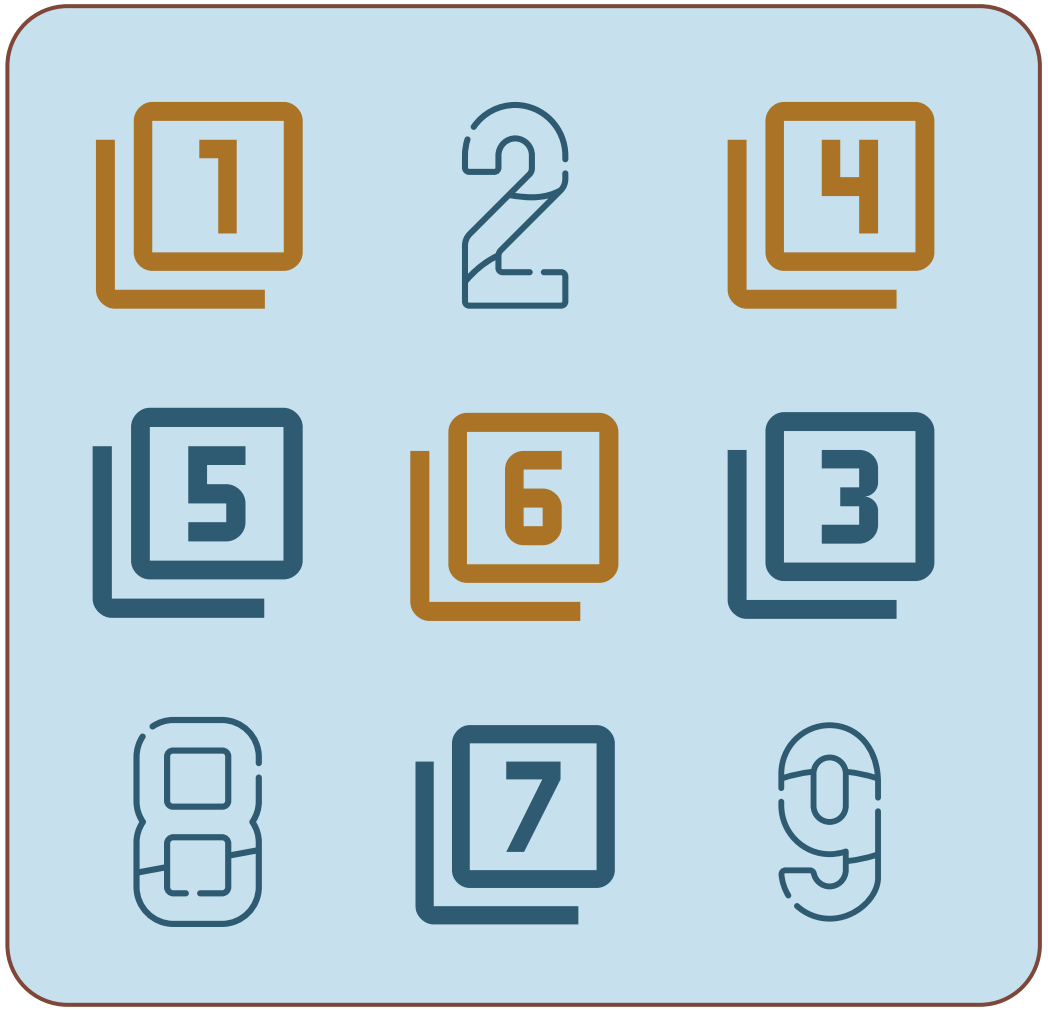


{C++ STORIES}

20 Smaller yet Handy C++20 Features

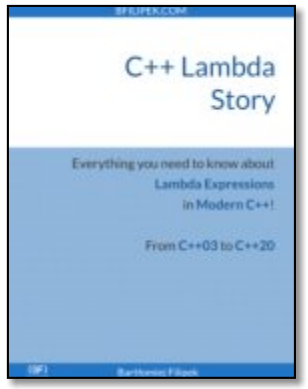
Part 1 - language

Bartłomiej Filipek, 29th September 2022, cppstories.com

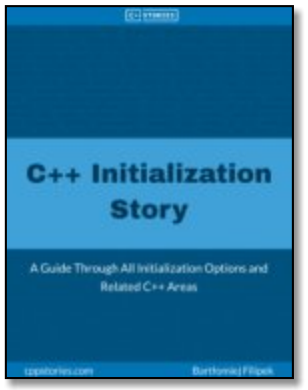
About Me

- Author of cppstories.com
- ~15y professional coding experience
- 4x Microsoft MVP, since 2018
- C++ ISO Member
- @Xara.com since 2014
  - Mostly text related features for advanced document editors
- Somehow addicted to C++ ☺

C++17 In Detail



C++ Lambda Story



C++ Initialization Story

Xara Cloud Demo

The plan

- About C++20
- 10 Language Features - today
- *10 Library Features - next part*
- *More in the future - next part*

About C++20

- 80 Library features and 70 language changes
  - https://en.cppreference.com/w/cpp/compiler_support#cpp20

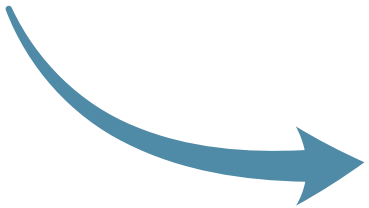
- Do you use C++20?
- Have you tried
  - `modules`
  - `std::format`
  - `concepts`
  - `coroutines`
  - `extended std::chrono?`

1. Abbreviated Function Templates and Constrained Auto

```
void myTemplateFunc(auto param) { }
```

```
template <typename T> void myTemplateFunc(T param) { }
```

```
template <class T>concept SignedIntegral = std::is_signed_v<T> && std::is_integral_v<T>;

void signedIntsOnly(SignedIntegral auto val) { }
void floatsOnly(std::floating_point auto fp) { }

// above is equivalent to:
template <class T>concept SignedIntegral = std::is_signed_v<T> && std::is_integral_v<T>;
template <SignedIntegral T> void signedIntsOnly(T val) { }
template <std::floating_point T>void floatsOnly(T fp) { }
```



```
template <typename T> requires SignedIntegral<T>
void signedIntsOnly(T val) { }
template <typename T> requires std::floating_point<T>
void floatsOnly(T fp) { }
```

2. Template Syntax For Generic Lambdas

```
const auto fooDouble = [](auto x, auto y) { /*...*/ };
```

```
struct {
    template<typename T, typename U>
    void operator()(T x, U y) const {
        /*...*/
    }
} someOtherInstance;
```

```
// C++17
auto ForwardToTestFunc = [](auto&& ...args) {
  // what's the type of `args` ?
  return TestFunc(std::forward<decltype(args)>(args)..
};
```

```
auto fn = []<typename T>(vector<T> const& vec) {
    cout << size(vec) << “, “ << vec.capacity();
};
```

```
auto GenLambda = [](std::signed_integral auto param) {
    return param * param + 1;
};
```

3. Constexpr Improvements

- union - P1330
- try and catch - P1002
- dynamic_cast and typeid - P1327
- constexpr allocation P0784
- Virtual calls in constant expressions P1064
- Miscellaneous constexpr library bits... later...

4. using enum

```
enum class long_enum_name { hello, world, coding };
void func(long_enum_name len) {
#if defined(__cpp_using_enum) // c++20 feature testing
    switch (len) {
        using enum long_enum_name;
        case hello: std::cout << "hello "; break;
        case world: std::cout << "world "; break;
        case coding: std::cout << "coding "; break;
    }
#else
    switch (len) {
        case long_enum_name::hello: std::cout << "hello "; break;
        case long_enum_name::world: std::cout << "world "; break;
        case long_enum_name::coding: std::cout << "coding "; break;
    }
#endif
}
```

5. Class-types in non-type template parameters (NTTP)

- Before C++20, for a non type template parameter, you could use:
  - lvalue reference type (to object or to function);
  - an integral type;
  - a pointer type (to object or to function);
  - a pointer to member type (to member object or to member function);
  - an enumeration type;
- But since C++20, we can now add:
  - structures and simple classes - structural types
  - floating-point numbers
  - lambdas

Basic: https://godbolt.org/z/a9718GMqz
From the proposal, string literal wrapper: https://godbolt.org/z/e54E4v69r

6. constinit

```
#include <string>
#include <iostream>

// init at compile time
int x;                  // zero initialization, don't use!
int y = 10;             // constant initialization
std::string compute() { return "hi\n"; }
constinit std::string global = {"hello\n"};
// constinit std::string global = compute(); // error

int main() {
    std::cout << global;
    // but allow to change later...
    global = "abc";
    std::cout << global;
}
```

https://godbolt.org/z/3eWsvhfdn

7. Designated Initializers

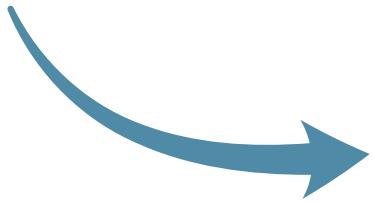
```
struct Date {int year;int month;int day;};

// easier to read:
Date inFuture { .year = 2050, .month = 4, .day = 10 };
// than:
Date inFuture { 2050, 4, 10 };
```

- Here are the main rules of this feature:
  - Only for aggregate types and for non-static data members
  - They have to have the same order of data members in a class declaration (not in C)
  - Not all data members must be specified in the expression
  - You cannot mix regular initialization with designers
  - There can only be one designator for a data member
  - You cannot nest designators.

8. Nodiscard Attribute Improvements

```
[[nodiscard("Don't call this heavy function if you don't need the result!")]]
bool Compute();
```

try: https://godbolt.org/z/16Kzbse8z



```
<source>: In function 'int main()':
<source>:9:16: warning: ignoring return value of 'bool Compute()', declared with attribute
'nodiscard': 'Don't call this heavy function if you don't need the result!' [-Wunused-result]
    9 |         Compute();
      |         ~~~~~~~~^~
```

What's more thanks to P0600 this attribute is now applied in many places in the Standard Library, for example:

- `async()`
- `allocate(), operator new`
- `launder(), empty()` (see example at https://godbolt.org/z/j3bcjYYMn)

9. Range-based for loop with Initializer

```
void print(const std::ranges::range auto& container) {
    for (std::size_t i = 0; const auto& x : container) {
        std::cout << i << " -> " << x << '\n';
        // or std::cout << std::format("{} -> {}", i, x);
        ++i;
    }
}

// undefined behavior if foo() returns by value
for (auto& x : foo().items()) { /* .. */ }

// fine:
for (T thing = foo(); auto& x : thing.items()) { /* … */ }
```

10. New keyword consteval - immediate functions

```
consteval int sum(int a, int b) { return a + b; }
constexpr int sum_c(int a, int b) { return a + b; }

int main() {
    constexpr auto c = sum(100, 100);
    static_assert(c == 200);

    constexpr auto val = 10;
    static_assert(sum(val, val) == 2*val);

    int a = 10;
    int b = sum_c(a, 10); // fine with constexpr function

    // int d = sum(a, 10); // error! the value of 'a' is
                           // not usable in a constant expression
}

    // consteval int some_important_constant = 42; // error
```

And more!

- List of supported features: https://en.cppreference.com/w/cpp/compiler_support#cpp20
- C++20 - The Complete Guide, by N Josuttis - https://leanpub.com/cpp20
- Google Chrome: C++20, How Hard Could It Be - presentation and discussion on Reddit: https://www.reddit.com/r/cpp/comments/xnk3fm/google_chrome_c20_how_hard_could_it_be/
- My articles on C++20: https://www.cppstories.com/tags/cpp20/

Summary

Abbreviated Function Templates and Constrained Auto
Template Syntax For Generic Lambdas
Constexpr Improvements
using enum
Class-types in non-type template parameters
New keyword constinit
Designated Initializers
Nodiscard Attribute Improvements
Range-based for loop with Initializer
New keyword consteval - immediate functions