

1. Cost of using static

```
struct c
{
    static const std::string getInstance()
    {
        static std::string str = "abc";
        return str;
    }
};
```

// In this ex. it is will always check if str is initialized or not always when we call
GetInstance() method,

// To avoid if check

```
struct c
{
    static const void CreateInstance()
    {
        static std::string str = "abc";
    }
    static const std::string& getInstanceRef()
    {
        return str;
    }
};
```

use getInstanceRef() directly

2. Clang-tidy is free opensource static code analysis tool. It can also fix the warning/error in code using -fix-errors .

3. std::endl is new line with flush, use '\n' instead

4. Don't write in loop on fostream object, instead write it in stringstream first and then at end write it in file ostream. This will save lots of file access.

5. use std::stringstream ss; ss << '\n' instead of expensive string version ss << "\n". In string version it will tru to iterate over string obj and new string object will created.

6. In function call, the gcc evaluate parameters from right to left and cland from left to right, so standard does not specify the order for evaluation

7. If you want to guarrenty the order of execution then use initializer list

ex.

```
template<typename ...T>
void print(const T& ... t) {
    std::initializer_list<int>{ (print_impl(t), 0)... };
}
```

```
}  
print(f1(), f2());
```

or

```
(void)std::initializer_list<int>{ (f1(), f2()) };
```

or

```
int i = (f1(), f2())
```

8. In `std::bind`, we bind the function with parameters, if we want to transfer the ref, then we have to use `std::ref()` explicitly.

suppose we want to bind one parameter and another user should provide then we have to use `std::placeholders::_1`:

ex. `auto f = std::bind(&print, std::ref(i), std::placeholders::_1);` // `_1` is position of parameter

`bind` designed to allow extra arguments provided

we can change the position of parameter but still will be passed correctly may be based on matching data type

```
auto f = std::bind(&print, std::placeholders::_2, std::placeholders::_1);  
f(4, "abc");
```

9. `lambda` are more efficient than `std::bind`, avoid using `std::bind`, it has both compile and runtime overhead

10. `std::invoke` is used to call functions with passing parameters. This gives us universal uniform interface/syntax.

No need to define function pointer for each type.

ex. `std::invoke(&something, 5);`

it is also used to access the data member, ex. `j` is public data member

`S s;`

```
std::invoke(&S::j, s)
```

11. `constexpr if`, header `type_traits`

It is used to deduce the return type of function at compile time.

ex.

```
template <typename T>  
auto print(const T&t)  
{  
    if(std::is_integer<T>::value)  
    {  
        return t+1;  
    }  
}
```

this will not compile, and will not be able to deduce return type.

With the addition of constexpr, it will compile

```
template <typename T>
auto print(const T&t)
{
    if constexpr(std::is_integer<T>::value)
    {
        return t+1;
    }
}
```

use case is if we want to use same function body with multiple types of return values.

12. Fold expression can be expanded either on left or right or in middle, ex. usually done for divide or multiply operator.

```
template<typename ...T>
auto x(T... t)
{
    return ( ... + t );
    return ( + ... t );
    return ( t + ... );
    return ( t + ... ) / sizeof...(t);
}
```

13. c++17 support the variable declaration in if/switch statement block , to reduce life cycle of variable.

```
if(int x =0; x)
```

14. we can replace platform identification ie. #if defined(_XXX_) using

```
#if __has_include(<window.h>)
    #include <window.h>
#endif
```

15. auto [i, j] = S(); this is struct binding

16. structural aggregate initializer, instead of giving constructor, we can give curly bracket to initialize the members of structure.

```
ex. S s{12, 23};
```

S s{{12}, 23, 3.4}; // this is for when S is derived from a base class having one int member and S is having 2 data members.

17. C++ 17 removed register keyword, as modern compiler does not require it. Also, boolean increment is removed. Random_shuffle is also removed, as it was broken.

also, std::bind1st is removed.

18. `c++17` changes in sequence containers. worth noting, `emplace_back` and `emplace_front` return ref to object added

```
auto ref = a.emplace_back(23);
```

19. `[[fall_through]]` attribute : this attribute used in switch statements.

```
switch (args)
{
    case 1:
        do_something();
        [[fall_through]]; // when case 1 execute it will execute case 2 also, this
way we can execute case 2 separately
    case 2:
        do_somethingelse();
        break;
}
```

20. `[[maybe_unused]]` : mark things if unused, like args to functions, local variable, functions etc.

```
[[maybe_unused]] int x, [[maybe_unused]] int func( int [[maybe_unused]])
```

21. `[[nodiscard]]` : it can be applied to function or class or structure or enum, to prevent a leak or error code

22. `#include <os>`

direct os api, more on hardware interface.

23. cost of using Lambdas

invoking lambda in same line, return `[]{ return 5;}();`

1. if we not enable optimisation then it will generate same amount of code as struct with `()` operator, when we enable optimisation very much

24. `-Og` option will enable optimisation that will not come in debug path.

25. Constructor initializer vs Default member initializer

```
struct S
{
    S() : value(5) {}
    int value;
};
```

```
struct S
{
    int value = 5; //Default member initializer
}
```

if we have 2 choices, then second one is best for generating less number of code and more efficient code.

26. Stateful Lambdas

the lambda object in which we have some data member/static data/data on heap.

ex. `auto l = [i = 0, j = std::make_unique()]() mutable { return ++i;}`

fibonacci using lambda

```
auto l = [i = 0, j = 1]() mutable {
    i = std::exchange(j, i+j);
    return i;
}
```

```
for (int i = 0; i < 10 : ++i) { l(); }
```

27. C++ 17, deduction guide.

we can guide the compiler take a look for type by specifying the type in std namespace

```
namespace std
{
    template<typename Ret, typename ... Arg> function(ret (* (Arg ...)) -> function
<Ret (Arg ..))
}

void test() {}

int main () {
    std::function f(&test);
}
```

28. Inheriting from Lambdas

we can pass lambda as template to class, for this we need a helper function and decay_t to cast it to type.

29. C++17 has variadic declaration syntax for inheriting

```
template<typename ...B>
struct Merge : B...
{
    template<typename ...T>
    Merge(T&& ... t) : B(std::forward<T>(t))...
    {
```

```

    }
    using B::operator()...;
}

```

```

template <typename ...T>
Merge (T...) -> Merge <std::decay_t<T> ...>;

```

```

int main()
{
    std::array<std::variant<double, int, char, std::string>, 3> a (3.2, 3, "Hello");
    int totalInt = 0;
    double totalDouble = 0;

    Merge m{ [&totalInt](int i){intTotal +=i;},
             [&doubleTotal] (const double d) { doubleTotal += d;}
             [](const std::string&) {},
             [](const char c){}
             };

    for_each(begin(a), end(a),
             [&m] (const auto &v) {std::visit(m, v); });
}

```

30. -0xFFFFFFFF is 1, as we negation applied to unsigned value only give unsigned value in result

31. Lambdas in which we define object it become stateful.

32. no op, no operation can be added by using ';' for loop internally implemented as while 'continue' is nothing but 'goto'

33. std::quoted, it is a function which add quotes around the string

```

std::quoted("Hello")
op: "Hello"

```

```

std::quote_escape()

```

34. std::cout << -1.0 * 0.0 ;
op/ = -0.0

`std::signbit()` // return if negative value

`std::copysign(,)` // copy sign from rhs to lhs

35. `std::fmin()` // this is for floating point comparison

`std::fmin(NAN, 1.9);` // NAN is empty placeholder non number value

36. If we don't have c++17 fold expression support then we can use `std::initializer_list` instead of it

37. `std::gcd` and `std::lcm` are newly added in c++ 17, header is `<numeric>`

38. `std::iota`, it will initialize the container with incrementing value

39. `std::search` will help in searching subset within other container

40. `quick-bench.com`, can be used to benchmark the c++ code.

41. c++17 has, `static_pointer_cast`, `dynamic_pointer_cast`, `const_pointer_cast`, `reinterpret_pointer_cast`, basically used on shared pointer.

42. `std::to_char`, convert the value to string, also there is `std::from_char` function

43. used `shared_timed_mutex` (which will timed out) over `std::shared_mutex` to avoid infinite time waiting

44. `string_view` only hold pointer to start and end of string, does not allocate any memory

45. we cannot move from const object,

`const S s;`

`S s2(std::move(s));` // this will only work when there is s is non const,

// it will not work with `S(&&)`, it will call copy constructor instead.

// Note: `S(const &&)` is not possible, is invalid move constructor

46. `static_print` , to print at compile time

`static_print(decltype(foo()));` // op can be int float double etc.

47. compiler explorer can be setup on local machine

48. `clang` and `gcc` support, thread, address, memory, undefined behavior, data flow, leak, coverage, stats etc. sanitizer built in.

In unit test framework we should add such flags which add instrumentation

`-fsanitize=address`

`-fsanitize=thread`

-fsanitize=leak
-fsanitize=undefined

49. Fuzz testing , we can use american pol fuzz testing tool to generate random string for testing input

50. `std::optional`, it is used in case when you don't want to create memory for huge object until it is required or exists,

with `std::optional` it does not create heap memory until something is assigned to it.

51. by adding `iostream` we add some global variables like `cout` `cin` etc.

52. `std::puts`, this will print the string on standard output and puts new line, this is efficient than use of `std::cout` (to avoid global variable)

53. overusing the lambda will increase the binary size.

54. Lippincott Functions: centralized function for catching all exceptions

```
ex. catch (...) { Lippincott(); }  
void Lippincott() {  
    try {  
        throw ;  
    } catch (const std::runtime_error&) {}  
    } catch (const std::exception&) {}  
}
```

55. function-try-block

```
func()  
{  
    try {  
    } catch (...)  
    {  
  
    }  
}
```

this is valid syntax for function, can one use case is

```
struct S {  
    S(int x) try  
        : tt(val)  
        {} catch (int) {}  
  
    ThrowingThing tt;  
};
```


56. exception cannot be consumed when function-try-block is used in constructor call, it will always rethrow it, there is no return statement allowed in catch block of constructor's function-try-catch.

57. custom comparator can be operator < or operator(), we call it as predicate of functor. and last is lambda so 3 ways.

std does not allow lambda inside template parameter, we need to pass type of lambda. we have to use decltype()

```
const auto lambda = [](){};  
std::set<Mydate, decltype(lambda)> myset{lambda}; // note: we need to pass the lambda  
obj in constructor of set.
```

58. Transparent comparator, comparator for ==

```
struct Mydate  
{  
    string key;  
};  
  
struct compare  
{  
    bool operator() (const Mydate& lhs, const Mydata& rhs) const {  
        return lhs.key < rhs.key;  
    }  
  
    template<typename T>  
    bool operator() (const T& lhs, const Mydata& rhs) const {  
        return lhs < rhs.key;  
    }  
  
    template<typename T>  
    bool operator() (const Mydate& lhs, const T& rhs) const {  
        return lhs.key < rhs;  
    }  
};  
  
int main()  
{  
    set<Mydate, compare> myset;  
    myset.insert(Mydata("Bob"));  
    return myset.count("Bob") > 0;  
}
```

59. std::decay_t , Performs the type conversions equivalent to the ones performed when passing function arguments by value.

```
template< class T >
using decay_t = typename decay<T>::type;
```

60. Transparent comparator using lambda

```
template<typename T, typename ... Comparator>
auto make_set(Comparator && ... comparator)
{
    struct Compare : Comparator...
    {
        using Comparator::operator()...;
        using is_transparent = int;
    };

    return std::set<T,
Compare>{Compare{std::forward<Comparator>(comparator)...}};

}

struct Person
{
    std::string name;
};

int main()
{
    auto set = make_set<Person>(
        [](const Person &lhs, const Person &rhs){ return lhs.name < rhs.name; },
        [](const Person &lhs, const auto &rhs){ return lhs.name < rhs; },
        [](const auto &lhs, const Person &rhs){ return lhs < rhs.name; }
    );

    set.insert(Person{"Bob"});
    return set.count("Bob");
}
```

61. lambda can be stored in function pointers. and fp can be used to in container declaration where we pass comparator as fp.

62. `std::accumulate(std::begin(value), std::end(value), 0);` // this will return int value
`std::accumulate(std::begin(value), std::end(value), 0.0);` // this will return float value

63. Bit-field
 struct S

```

{
    int i : 3; // here size of i is 3 bits and not 32 bits
    int j : 5; // here size of j is 5 bits and not 32 bits
};

```

size of S, should/can be 8 bits (depend on compiler) or mostly 32 bits (24 bits added for padding alignment).

64. All assignment operator versions, total 64 possibles, some of them are by default nto allowed but those can have custom impl

```

struct S
{
    S &operator=(const S &) &= default; // l value ref overload, these can be
expand to const variant
    S &operator=(S &&) &= default;
    S &operator=(S &) &= default;
    S &operator=(const S &&) &= delete;

    S &operator=(const S &) &&= default; // r value ref overload, these can be
expand to const variant
    S &operator=(S &&) &&= default; // S{} = S{}; possible with this operator
    S &operator=(S &) &&= default;
    S &operator=(const S &&) &&= delete;

    S &operator=(const S &) const &= delete; // l value ref overload, these can
be expand to const variant
    S &operator=(S &&) const &= delete;
    S &operator=(S &) const &= delete;
    S &operator=(const S &&) const &= delete;

    S &operator=(const S &) const &&= delete; // l value ref overload, these can
be expand to const variant
    S &operator=(S &&) const &&= delete;
    S &operator=(S &) const &&= delete;
    S &operator=(const S &&) const &&= delete;
};

```

65. <stdio.h> is depricated we should use now <cstdio> onwards

66. visual studio clang tidy extension for formating and removing warnings automatically. Also, cpp check addon available

67. Disable move from const object

`S(const S&&) = delete; // compiler does not allow move from const instead it silently do copy, this we are deleting copy from temp object`

`S &operator=(const S &&) = delete; // compiler does not allow move from const instead it silently do copy, this we are deleting copy from temp object, indirectly disabling move from const.`

68. delete can be used to explicitly disabling implicit type converison and calling overload, this can be applied for template and non template function.

ex.

`void foo(int);`

`void foo(double) = delete; // this will restrict type conversion`

69. `push_back(S());` // below operator will get called

`S(), S(&&), ~S(), ~S()`

`emplace_back();`

`S(), ~S()` no temporary object

70. When noexcept Really Matters

ex. when vector reaches it size and it need to reallocate new mem for expansion then if noexcept is not provided to move constructor or operator then it will call copy objects, if move constructor is provided with noexcept then it will do move of objects.

Basically, stl uses move of objects in their algo when noexcept guarrenty is provided on move constructor.

71. GCC's Leaky Abstractions

`auto l = [val = 2]() { return val; }`

`l. __val = 4;`

72. Similar to QT, there are other UI lib like gtkmm, nana, fltk and imgui

73. Conan package manager, is to install the c++ libs. available on linux.

74. The Hunter Package Manager, vcpkg Package Manager

75. cost of number of args to function

as number of args increase, the compiler uses more number of registers, ex. for 5 args compiler use 5 registers

so to optimize this we can use struct and pass all args to function, this will use only one reg.

when number of regsiter exhausted, it will use stack (push pop stack).

76. Strict Aliasing In The Real world

Aliasing refers to the situation where the same memory location can be accessed using different names

GCC compiler makes an assumption that pointers of different types will never point to the same memory location

This will affect the performance of code, if compiler is confused in 2 pointer can overlap then it will create 2 types of code(non overlapping and for overlapping bit by bit copy code), one is for overlap case and another is non overlap case, if we use strict types for 2 pointers then it will assume those will not overlap each other and will generate efficient (one case) code only.

ex. void foo(const std::uint8_t *src, std::uint32_t *dst);

here, src and dst can overlap each other as they are convertible (src can point to dst mem address also, its subset of dst)

but, void foo(const Data *src, std::uint32_t *dst);

here, compiler get guaranteed that it will not overlap each other, so code is generated accordingly.

char ptr consider to overlap to any data pointer,

-fno-strict-aliasing (and -fstrict-aliasing)

So basically if you have an int* pointing to some memory containing an int and then you point a float* to that memory and use it as a float you break the rule. If your code does not respect this, then the compiler's optimizer will most likely break your code.

The exception to the rule is a char*, which is allowed to point to any type.

77. std::in_place_t type of constructor

this will give on constructor for creating object in place, inplace is better than move

ex. std::optional<std::vector<int>> o(std::in_place, 10, 3); // create in place vector with value 10 & 3

78. ABM and BMI Instruction Sets

79. The Optimal Way To Return From A Function

in case of single or multiple return place, return temp value RVO or NRVO, return exact type what is require, type conversions are inefficient.

80. Lambdas With Destructors

we can create lambda inside lambda inside a capture block

ex.

auto l = [int i = 5]{ return i; }; // creating int

auto l = [i = []{ return 5; }{ return i; }]; // creating lambda which return 5 in capture block returning lambda from outer lambda

```
auto l = [i = []{
    struct S{
```

```

        int val = 5;
        S() { puts("const"); }
        ~S() { puts("dest"); }
    };
    return S{}; // returning S
;){ return i;}; // returning lambda which create S

```

81. Designated initializer c++20 feature

```

struct S
{
    int i;
    float f; // default value can be given as float f{1.0}; or float f{} i.e 0.0;
    double d;
};

S s{i:1, f:3.4};
S s{i:1, d:4.4};

```

82. Generic lambda

which is templatd lambda where typename is supported in c++20 and auto is supported in c++11.

```

auto l = [](auto a, auto b) {
    return a + b ;
};

auto l = []<template T>(T a, T b) {
    return a + b ;
};

```

83. for (auto x : get_s().get_data())

this will not work, as get_s is returning temp value and we are taking data obj from temp obj, hence it will give undefined behaviour.

84. lambda in fold expression

```

template < typename ... T>
auto do(T ...t)
{
    return ( [t]() { return t; } () + ...);
}

```

85. Get typeid

`typeid(l).name();`

86. `std::cerr` is an unbuffered version of `std::cout`, use it when you want to output immediately to flush.

87. `__attribute__((always_inline))`
`[[gnu::always_inline]]`
this attribute will always inline the function.

88. `[[gnu::pure]]`
`[[gnu::const]]`
many other flags which help in optimisation

89. VS code analysis tool gives a lot of suggestion on improvements.

90. `std::to_address`
utility gives address of object.

91. In C++ 20, we can create new lambda from previously defined lambda type.
and can be used to default constructor,
`std::set<data, decltype([](...){})> myset;`

92. `distcc` and `icecc` are used for distributed gcc build system,
`ccache`
C++ Weekly - Ep 153 - 24-Core C++ Builds Using Spare Computers!

93. Never define empty destructor, this is for rule of 0, and optimisation

94. Never overload operator `&&` or `||`, because it will not have ability of short circuit.
VIMP

95. Getting The Most Out Of Your CPU
specify `-mtune` or `-march` gcc compiler options

96. C++20's `is_constant_evaluated()`, to check if compile time evaluate

97. C++20's Uniform Container Erasure

98. `wshadow` used to highlight warning due to shadowing variables

99. inline static const should be replaced with static `std::string_view` `s()` { return "Hello world"; }
this has 0 runtime cost compared to static one which has to check for multithread
initializer

100. C++20's `std::bind_front`
`int add(int i, int j, int k);`

`auto func = std::bind_front(&add, 3, 2);`
`func(argc);` // 3 and 2 are passed as first 2 parameters

101. Spaceships in C++20: operator `<=>`
`#include <compare>`

`auto operator <=> (const Data& rhs) const = default;`

compiler will generate the `< = >` operator code for member wise data

102. c++ 20 support `std::bitcast`, which is used to casting from char bytes to object

103. C++20's `constexpr` `new` Support

`new constexpr` is supported when allocation and deallocation should be in same function.

or allocation should match the deallocation in flow. and allocation and deallocation should happen only in

`constexpr` context or function

104. variadic templates can be used to defined below

`template<typename T> constexpr bool is_same<T, T> = true;`

`template<typename T, typename U> constexpr bool is_same = false;`

105. Codeloop is free opensource IDE for c++

106. Stop Using Double Underscores

__ anywhere in the name

_P , underscore followed by capital letter

_m, in global section, stop using underscore followed by small letter

above mentioned cases are reserved for compiler usage, this can lead to undefined behavior

107. `_t` is for `::type` and `_v` is for `::value` from type trait.

108. SFINAE can be replaced with concept in c++ 20

109. `constexpr` forces compiler to init the local static variable at compile time.

use case is `std::mutex`

108. requires requires


```
c++20 has requires
requires requires { &FunctionObj::Operator(); }
```

or

```
template<typename T> concept
has_call_operator = requires {
    &T::operator();
};
requires has_call_operator<FunctionObj>
```

109. **Explicit constructor, used to create prevent accidently creation of expensive object.**

```
struct S {
    S(int) {}
    explicit S(float) {}
};

void f(const S&);
void g(const S&);

int main()
{
    f(10); // create S object by calling S(int) constructor
    g(10.9); // compilation error, cannot invoke S(float) constructor
    g(S(10.9)); //this will work
}
```

110. **prevent temporary creation**, obj which are expensive to create or hold some resources

```
struct S {
    [[nodiscard]] S(int);
};

{10}; // error
S(10); // error
```

111. [EP 201] For converting lambda to function pointer it should be non capturing lambda.

112. Note the bit manipulation features added in C++20

Standard library header `<bit>`

This header is part of the `numeric` library.

Functions

<code>bit_cast</code> (C++20)	reinterpret the object representation of one type as that of another (function template)
<code>ispow2</code> (C++20)	checks if a number is an integral power of two (function template)
<code>ceil2</code> (C++20)	finds the smallest integral power of two not less than the given value (function template)
<code>floor2</code> (C++20)	finds the largest integral power of two not greater than the given value (function template)
<code>log2p1</code> (C++20)	finds the smallest number of bits needed to represent the given value (function template)
<code>rotr</code> (C++20)	computes the result of bitwise left-rotation (function template)
<code>rotr</code> (C++20)	computes the result of bitwise right-rotation (function template)
<code>countl_zero</code> (C++20)	counts the number of consecutive 0 bits, starting from the most significant bit (function template)
<code>countl_one</code> (C++20)	counts the number of consecutive 1 bits, starting from the most significant bit (function template)
<code>countr_zero</code> (C++20)	counts the number of consecutive 0 bits, starting from the least significant bit (function template)
<code>countr_one</code> (C++20)	counts the number of consecutive 1 bits, starting from the least significant bit (function template)
<code>popcount</code> (C++20)	counts the number of 1 bits in an unsigned integer (function template)

113. EP 213, `CTRE: Compile time regular expression`

<https://github.com/hanickadot/compile-time-regular-expressions>

Use this instead `std::regex`, this will generate very less number of lines of code compared to standard code

114. PMR Example

```
#include <vector>
#include <memory_resource>

int main()
{
    std::byte stackBuf[2048];
    std::pmr::monotonic_buffer_resource rsrc(stackBuf, sizeof stackBuf);
    std::pmr::vector<int> vectorOfThings{{1,2,3,4,5,6}, &rsrc};
}
```

115. `Std::nextafter` returns the next value of provided one in the given direction. I.e to less than or more than current value. We can obtain next or previous smallest float value of given input.

116. We can have multiple destructor as shown below, if it matches the constrain then will pick it or else it will pick the default one

```

template<typename Contained>
struct Optional
{
    union { Contained data; };
    bool initialized = false;

    constexpr Optional &operator=(Contained &&data) {
        this->data = std::move(data);
        initialized = true;
        return *this;
    }

    constexpr ~Optional() requires(!std::is_trivially_destructible_v<Contained>)
    {
        if (initialized) {
            data.~Contained();
        }
    }

    constexpr ~Optional() = default;
};

```

117. Using enum,

This will save us from using scope resolution operator, it will bring the elements of enum in scope

118. In map using at() is more efficient than [] operators, as [] operator try to insert the default value if not available which is not seen in other kind of containers.

119. For PMR allocator use the correct allocator which is aware of container, ex.

```

77     std::array<std::uint8_t, 256> buffer{};
78     std::pmr::monotonic_buffer_resource mem_resource(buffer.data(),
79                                                     buffer.size());
80     print_buffer("initial", buffer, "");
81     std::pmr::vector<std::pmr::string> vec1({"Hello World", "Hello World 2", "Hel
82     print_buffer("2 short strings + 1 long string", buffer, vec1);
83 }
84
85 > void nested_long_string() {~
101 }
102
103 > void nested_long_pmr_string() {~
119 }
120
121 > struct S {~
138 };
139
140 > void aa_type() {~
163 }
164
165 int main() {
166     nested_string();
167
168     // * know how and if your memory is growing
169     // * choose the correct allocator for your use case
170     // * make sure you properly nest your allocator aware types
171     // * Understand which std types are allocator aware
172 }

```

Std::pmr::vector is aware of size multiplication behavior of vector and copying existing element while expanding allocation size, if we use incorrect allocator then it will duplicate the data and doesn't free the memory.

Also for string container, as it is Small object optimisation, it will allocate the memory on heap we don't use std::pmr::string allocator with string container.

120. In `[[nodiscard]]` we can provide additional message which will be printed in compilation time

```

1
2 struct Vector {
3     [[nodiscard("You probably meant to call clear()")]] bool empty() const;
4
5     void clear();
6 };

```

121. Ep 241 - Using `explicit` to Find Expensive Accidental Copies

Due to explicit copy constructor, it is preventing implicit copy in function call func(s)

```
struct S
{
    S() = default;
    explicit S(const S &);
};
```

```
void func(S);
```

```
int main()
{
    S s;

    func(s); // copy s in
}
```

122. Use of concepts + if constexpr, it check at compile time if capacity function is available or not.

```
auto allocated_size(const auto &container) {
    if constexpr (requires { container.capacity(); }) {
        return container.capacity();
    } else {
        return container.size();
    }
}
```

123. Below is an example where we are using print_alloc to detect any extra allocation done while using pmr resources.

// Prints if new/delete gets used.

```
class print_alloc : public std::pmr::memory_resource {
private:
    void* do_allocate(std::size_t bytes, std::size_t alignment) override {
        std::cout << "Allocating " << bytes << '\n';
        return std::pmr::new_delete_resource()->allocate(bytes, alignment);
    }
}
```

[illegible]

```

std::cout << "initializing vector\n";

//Note, container aware types
auto vec = create_container<std::pmr::vector<std::pmr::string>>(
    &mem_resource,
    "Hello", "World", "Hello Long String", "Another Long String");

std::cout << "exiting main\n";
}

```

124. Monotonic_buffer_resource (one large chunk of memory) allocates new memory block if exist one is full and it will deallocate it only when the monotonic_buffer_resource object is destroyed. It allocates different types or size of types in single continuous block

Unsynchronized_pool_resource

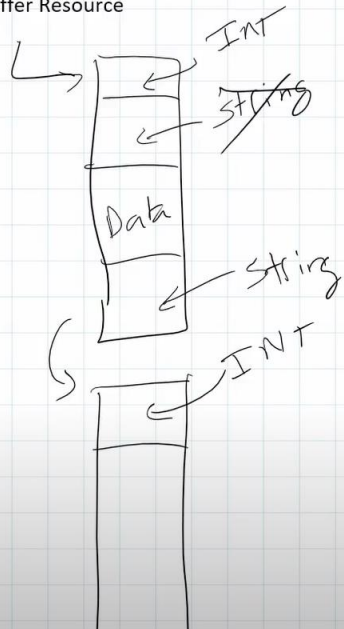
Synchronized_pool_resource

These are 2 other types for multi sized data types.

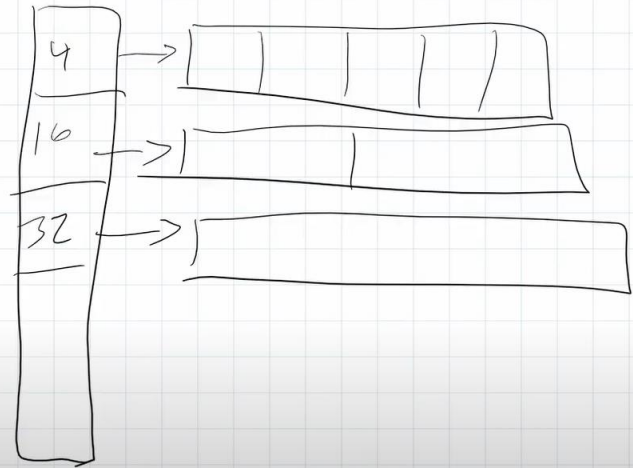
Link to examples: <https://godbolt.org/z/rvfofv>

polymorphic_allocator (C++17)	an allocator that supports run-time polymorphism based on the std::memory_resource it is constructed with (class template)
memory_resource (C++17)	an abstract interface for classes that encapsulate memory resources (class)
pool_options (C++17)	a set of constructor options for pool resources (class)
synchronized_pool_resource (C++17)	a thread-safe std::pmr::memory_resource for managing allocations in pools of different block sizes (class)
unsynchronized_pool_resource (C++17)	a thread-unsafe std::pmr::memory_resource for managing allocations in pools of different block sizes (class)
monotonic_buffer_resource (C++17)	a special-purpose std::pmr::memory_resource that releases the allocated memory only when the resource is destroyed (class)

Monotonic Buffer Resource

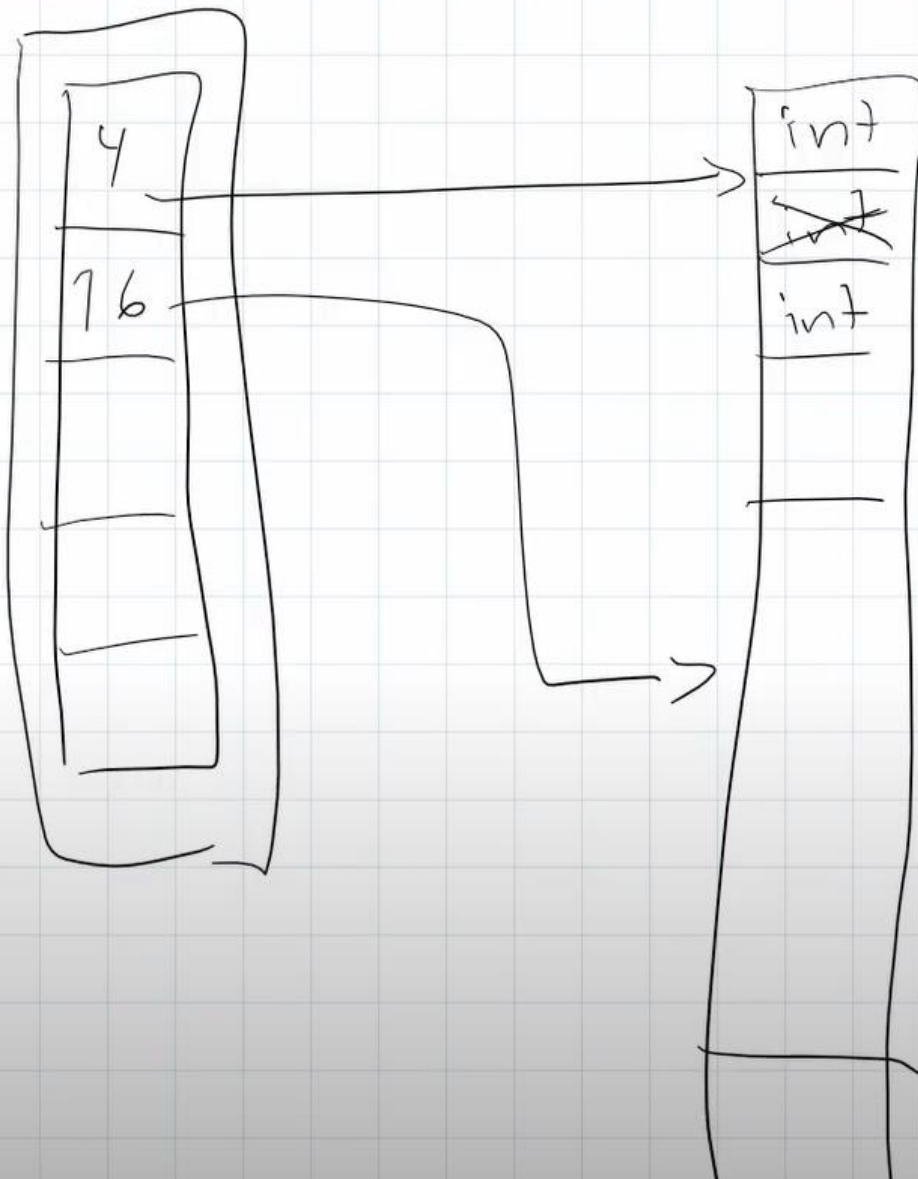


(un)synchronized Pool Resource



Unsynchronized pool resource combined with monotonic buffer resource.

(Un)synchronized Pool Resource -> Monotonic Buffer Resource



These resource object keeps the track of deleted object address so that next time it can be allocated to new request.

125. 3 unnamed types in C++

```

struct Outer {
    private:
        struct Inner { int i; };
    public:
        Inner get_inner() { return Inner(); }
};

```

```

auto inner_class()
{
    Outer o;
    auto val = o.get_inner();
    Outer::Inner val2 = o.get_inner();
}

```

```

auto local_class()
{
    struct MyStruct{};
    MyStruct s;
    return s;
}

```

```

auto lambda()
{
    return [](){};
}

```

1. Lambda
2. Inner private class
3. Class defined inside a function and return object of it

`std::vector<int, std::pmr::polymorphic_allocator>`

This allocator is using default memory resource whatever is set .

127. The Awesome Power of C++20's `std::source_location`

Return current line file column details function name

```
template<typename T>
void log(T t, std::source_location s = std::source_location::current())
{
    fmt::print("[{}] {}:({}, {}): {}\\n", s.file_name(), s.function_name(), s.line(), t);
}

int main(const int, const char * const *) {

    auto s = std::source_location::current();

    fmt::print("[{}] {}:({}, {})\n", s.file_name(), s.function_name(), s.line(),
    log("Hello World"/*, */);
}
```

128. EP 263, virtual inheritance

```

3  struct Base
4  {
5      int base_value = 42;
6  };
7
8  struct Intermediate1 : Base
9  {
10     int i = 1;
11 };
12
13 struct Intermediate2 : Base
14 {
15     int i = 2;
16 };
17
18 struct Derived : Intermediate1, Intermediate2
19 {
20
21 };
22
23 int main() {
24     Derived d;
25
26     ++d.Intermediate1::base_value;
27     --d.Intermediate2::base_value;
28
29     d.Intermediate1::base_value;
30     d.Intermediate2::base_value;

```

Here, line 29 return 43 and line 30 return 41, the object d has two values from base class, one is from Intermediate1 and another from Intermediate2, this is duplication of base object from two different derived paths, to avoid this we use virtual inheritance.

129. Covariant return type

```

1 struct Base
2 {
3     virtual Base *get_child() = 0;
4 };
5
6 struct Derived : Base
7 {
8     Derived *get_child() override {
9         return child;
10    }
11
12    Derived *child;
13 };

```

Here in derived class it is overridden the method and return type is not what is mentioned in base class method, but it still work as both class are in the hierarchy. **It will work for both return pointer and reference but not work for returning object.**

But unfortunately it does not work with smart pointers as compiler detect both types are different

```

4 struct Base
5 {
6     virtual std::shared_ptr<Base> get_child() = 0;
7 };
8
9 struct Derived : Base
10 {
11     std::shared_ptr<Derived> get_child() override {
12         assert(child);
13         return child;
14     }
15
16     std::shared_ptr<Derived> child;
17 };

```

130. Ep 266 - C++20's `std::shift_left` and `std::shift_right`

This is used to shift elements in **container**.

131. Ep 267 - C++20's `starts_with` and `ends_with`

Check if string start or end with particular set of character/s.

132. <https://cppinsights.io/>

Using this website we can check internal of c++ code generated by compiler.

<https://www.fluentcpp.com/>

It is the blogs on c++. Excellent contains

<https://devblogs.microsoft.com/oldnewthing/>

By Raymond chen

<https://www.youtube.com/@TheCherno/videos>

<https://www.youtube.com/@javidx9>

133. Ep 270 - Break ABI to Save C++

Return type is not part of name mangling this is to support the name mangling

If we change the order of virtual function definition in library then the order of entry in vtable for functions changes.

ABI Stability - What is Stuck

- Can never change the return type of a function
- Can never add/remove/reorder virtual functions
- Can never add/remove/reorder member variables
- Can never add/remove/reorder function parameters (but we can add new overloads, which mostly solves that issue)

ABI Stability - What is Potentially Lost

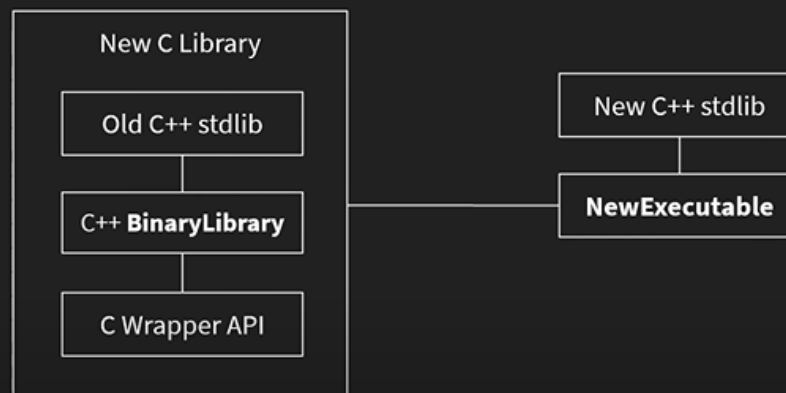
- Stdlib have a bug that would require changing layout? Sorry, we're stuck with that bug!
- Would adding/removing a member variable to `std::vector` increase performance or save bytes? Can't do it!
- Is there a bug in the design of the stdlib that needs to be fixed? Sorry, that's probably an ABI break!

<https://cor3ntin.github.io/posts/abi/>

Nice blogs on c++ <https://cor3ntin.github.io/>

To protect us from abi breakage use c wrapper on top of old compiled libs in c++

Providing a C Wrapper for Your **BinaryLibrary**



Note: This is kind of how Windows system DLLs are written

134. Use `string.clear()` rather than using `string = ""`;

Using `string.clear()` is 3 times faster than `= ""`;

135. Std::pair is basically struct with 2 members, first and second

```
template<typename First, typename Second>
struct Pair{
    First first;
    Second second;

    template<typename F, typename S>
    Pair(F &&f, S &&s)
        : first(std::forward<F>(f)), second(std::forward<S>(s))
    {
    }

    Pair(const First &f, const Second &s)
        : first(f), second(s)
    {
    }
};
```

136. Ep 276 - C++20's Conditionally `explicit` Conversions

```
template <typename Contained>
struct S {
    S() = default;
    explicit(!std::is_trivial_v<Contained>) S(Contained &&c_) : cont
    explicit(!std::is_trivial_v<Contained>) S(const Contained &c_) :
```

We are marking constructor as explicit to avoid implicit conversion as they are expensive, but it is not true for all types, some types are check or trivial for conversion implicitly, hence we are providing condition in explicit clause.

137. Avoid signed unsigned conversions that leads to degrade performance.

138. Tip for when to use `emplace_back` and `push_back`


```

std::vector<std::string> vec;

// call emplace_back when you want to create a new object
vec.emplace_back(100, 'c');           This is placement new

// 1. reserve space for a new string
// 2. placement new() into new space (args...)

// call push_back when you already have an object
vec.push_back(std::string(100, 'c'));

// 1. create a temporary string on the stack(100, 'c')
// 2. resize the vector
// 3. std::move(temporaryString) into new memory location

vec.emplace_back(std::string(100, 'c')); this is move constructor + placement new

// 0. create temporary string on the stack
// 1. reserve space for a new string
// 2. placement new() into new space (args...) (move constructor)

```

Note first version of `emplace_back` is more faster than second.

139. `!=` is more efficient than `<` incase of comparison,

140. Ep 283 - Stop Using `const_cast`!

`Declytype(auto)` is inverse of perfect forward, it will return exact type

142. Safe integer comparsion EP 284,

```

int getSValue() ; // return -1
int unsigned getUIValue() // return 1
bool res = getSValue() < getUIValue();

```

It will give the warning and result would be false, expected was true

If we use static cast around `getUIValue` then result will be true, as expected

```

bool res = getSValue() < static_cast<int>(getUIValue());

```

If we use `static_cast` around `getSValue` then result would be fasle, expected is true.

```

bool res = static_cast<unsigned int>(getSValue()) < getUIValue();

```

This is because

signed int represent -1 as 1111'1111 (2's compliment)

unsigned signed int represents 1 as 0000'0001

Hint: always use static cast around unsigned value rather than signed one

If we `static_cast<unsigned int>(-1)`; then it will give `max_int` ie. `1111'1111`
So compared to `0000'0001` it is larger which will give wrong result

In c++20 it has given utility function to solve this issue

Integer comparison functions

Defined in header `<utility>`

<code>cmp_equal</code>	
<code>cmp_not_equal</code>	
<code>cmp_less</code>	(C++20) compares two integer values without value change caused by conversion
<code>cmp_greater</code>	(function template)
<code>cmp_less_equal</code>	
<code>cmp_greater_equal</code>	
<code>in_range</code> (C++20)	checks if an integer value is in the range of a given integer type (function template)

143. Auto

```
// * `auto` uses the exact same rules as template type parameters
// * `auto` will never deduce a `&`
// * `const`-ness will be deduced
// * `auto` will never perform a conversion!
// * `auto` is generally better performing because it eliminates
//   accidental conversions!
```

144. Adding const ness

We can use `const_cast` to add the const ness to object

As we know `const_cast` is used to remove the constness of object on the other hand it can be used to add the const ness

Also, we can use the `static_cast` to add constness

Ex. `int i = 10;`

`static_cast <const int> (i);`

145. We can provide the constraint to auto in c++ 20, syntax is

Std::floating_point auto GetValue();

146. **abidiff** it is tool to see the differences of abi between two releases of binaries or header files or obj files.

```
jason jason-manjaro ~ abidiff > libabigail-1.8 > tools $ ./abidiff --redundant --verbose --stats --harmless ../../test1/test.o ../../test2/test.o | c++filt
Functions changes summary: 0 Removed, 0 Changed, 0 Added function
Variables changes summary: 0 Removed, 0 Changed, 0 Added variable
Function symbols changes summary: 1 Removed, 1 Added function symbols not referenced by debug info
Variable symbols changes summary: 0 Removed, 0 Added variable symbol not referenced by debug info

1 Removed function symbol not referenced by debug info:

[D] use_params(Params&, int)

1 Added function symbol not referenced by debug info:

[A] use_params(Params&)
```

<https://sourceware.org/libabigail/manual/abidiff.html>

Another tool “c++filt” is used to demangle the c++ names

https://sourceware.org/binutils/docs/binutils/c_002b_002bfilt.html

147. https://en.cppreference.com/w/cpp/types/is_constant_evaluated

Detects whether the function call occurs within a constant-evaluated context. Returns true if the evaluation of the call occurs within the evaluation of an expression or conversion that is manifestly constant-evaluated; otherwise returns false.

```
int y = 0;
```

```
const int a = std::is_constant_evaluated() ? y : 1;
```

```
// Trial constant evaluation fails. The constant evaluation is discarded.
```

```
// Variable a is dynamically initialized with 1
```

```
const int b = std::is_constant_evaluated() ? 2 : y;
```

```
// Constant evaluation with std::is_constant_evaluated() == true succeeds.
```

```
// Variable b is statically initialized with 2
```

148. Result will be 255

```
int main()
{
    std::uint8_t value1 = 0;
    std::uint8_t value2 = 1;
    std::uint8_t result = value1 - value2;
```

149. Note in case of signed int, the sign bit maintained irrespective of shift direction.

But this is not true for unsigned int

```
// signed integer arithmetic
// 11000000 >> 1
// 11100000
```

so always do shift operation on unsigned int only

```
// unsigned integer arithmetic
// 11000000 >> 1
// 01100000
```

150. constexpr

To force the compiler calculate at compile time use static assert on constexpr value, this way compiler will be forced to guaranteed evaluate this at compile time.

Also, constexpr evaluated are stays on stacks, in below example it is undefined behaviour as the value is init on stack and it goes away when scope is end, and the pointer p will have dangling pointer. solution is use **static constexpr**

```
int main() {  
    const int *p = nullptr;  
  
    {  
        constexpr auto values = get_values();  
        // constexpr values are stack values!  
        p = &values[985];  
    }  
  
    return *p;  
}
```

151.Ep 320 - Using `inline namespace` To Save Your ABI

Inline namespace can be used to generate versions for lib interfaces, to bring the namespace in effect without modifying code, we can inline namespace which ever we want to.

```

4 namespace lefticus {
5 namespace v2_0_0 {
6 struct Data {
7     int i;
8     char c;
9     char c2;
10 };
11
12 int calculate_things(const Data &);
13 } // namespace v3_0_0
14
15
16 // downside: we have to manually keep track of t
17 inline namespace v3_0_0 {
18 struct Data {
19     int i;
20     char c;
21     char c2;
22 };
23
24 int calculate_things(const Data &);
25 } // namespace v3_0_0
26 } // namespace lefticus
27
28 int main() {
29     const lefticus::Data some_data{};

```

152. Don't use const in below places, to gain move operation

```

// #4 don't `const` non-reference return types
inline const S make_value_4() // bad, non-& const return type!
{
    return S{};
}

// #3 don't `const` local values that need take advantage
//   of implicit move-on-return operations
inline std::optional<S> make_value_3() {
    const S s; // bad because this object and the return type
               // are different!
    return s;
}

```

This will break move operation and will use copy instead.

Better to use

`Const std::string s = make_value();`

In below code it will eliminate return value optimisation

```

// #3.1 if you have multiple different objects that might
//       be returned, then you are also relying on
//       implicit move-on-return (aka automatic move)
inline S make_value_3_1(bool option) {
    if (option) {
        const S s; // bad use of const
        return s;
    } else {
        const S s2; // bad use of const
        return s2; // better would be to return S{} directly!
    }
}

```

```

// #2 don't `const` non-trivial value parameters that you
// might need to return directly from the function
inline S make_value_2(const S s) // return statement makes this bad use of `const`
{
    return s; // because we return it, const is bad in function definition!
}

// #1 don't `const` *any* member data!
// it breaks implicit and explicit moves
// it breaks common use cases of assignment
// (you might need to write some accessors!)
struct Data
{
    const S s;
};

struct StringData
{
    const std::string s;
};

int main([[maybe_unused]] int argc, const char *[])
{
    std::vector<Data> data;
    data.emplace_back();
    data.emplace_back();
}

```

153. Correct use of std::erase

Dont use

std::erase(containter, container.front()); // this will give undefined behaviour as erase is taking last parameter by const& and then internally use remove_if algo, which swaps objects which are matching with criteria. In this process the front object might get overwritten with other object and we end up having wrong matching criteria.

Use copy of instead

std::erase(containter, auto{container.front()};);

154. Ep 324 - C++20's Feature Test Macros

Check compile time if compiler has that feature

__has_cpp_attribute(attribute-token)

https://en.cppreference.com/w/cpp/feature_test

155. Ep 326 - C++23's Deducing `this`

```
template <typename Contained, std::size_t Width, std::size_t Height>
struct Matrix {

    template<typename Self>
    auto &at(this Self &&self, std::size_t X, std::size_t Y ) {
        return std::forward<Self>(self).data.at(Y * Width + X);
    }
    std::array<Contained, Width * Height> data;
};

int main() {
    const Matrix<int, 5, 5> data{};

    using const_at = const int &(*)(const Matrix<int, 5, 5> &, std::size_t , std::size_t );

    auto func = Matrix<int, 5, 5>::at<const Matrix<int, 5, 5> &>;

    return func(data, 2, 3);
    //data.at(2, 3) = 15;
    //return data.at(2, 3);
}
```

156. Ep 327 - C++23's Multidimensional Subscript Operator Support

157. Ep 328 - Recursive Lambdas in C++23

```
int main() {
    constexpr auto factorial = []<typename Self>(this const Self &self,
                                                int value) {
        if (value == 1) {
            return value;
        } else {
            return value * self(value - 1);
        }
    };

    return factorial(4);
}
```

158. inter procedural optimization

Interprocedural optimization (IPO) is a collection of compiler techniques used in computer programming to improve performance in programs containing many frequently used functions of small or medium length. IPO differs from other compiler optimizations by analyzing the entire program as opposed to a single function or block of code.

IPO seeks to reduce or eliminate duplicate calculations and inefficient use of memory and to simplify iterative sequences such as loops. If a call to another routine occurs within a loop, IPO analysis may determine that it is best to inline that routine. Additionally, IPO may re-order the routines for better memory layout and locality.

IPO may also include typical compiler optimizations applied on a whole-program level, for example, dead code elimination (DCE), which removes code that is never executed. IPO also tries to ensure better use of constants. Modern compilers offer IPO as an option at compile-time. The actual IPO process may occur at any step between the human-readable source code and producing a finished executable binary program.

We have to enable explicitly option/flag for this to work.

159. Link Time Optimization (LTO) gives GCC the capability of dumping its internal representation (GIMPLE) to disk, so that all the different compilation units that make up a single executable can be optimized as a single module. This expands the scope of inter-procedural optimizations to encompass the whole program (or, rather, everything that is visible at link time).

160. Use of `extern template` to solve multiple definitions of a template function.

```
template<typename Type>
Type add(Type lhs, Type rhs) {
    return lhs + rhs;
}

//extern template saves us on build time in each
// cpp file of our large project
// (assuming this `add` function is expensive to compile!)
extern template int add<int>(int, int);
```

We can add this extern in file and start using add function, at link time it will link to definition.

161. We can convert the captureless lambda to function pointer.

162. Ep 334 - How to Put a Lambda in a Container

We can use `std::function` or function pointer, better to use function pointer (for lambda with no capture).


```

auto make_lambda(int value) {
    return [value](int i) { return i + value; };
}

int main()
{
    std::vector<decltype(make_lambda(42))> data;

    data.push_back(make_lambda(1));
    data.push_back(make_lambda(2));
}

```

This is another way where return type of make_lambda is always same so we can create vector of it and add new lambdas taking different values. But this has a problem of one definition rule problem, as every compilation unit will going to create new type. To solve this we have to add our own type. That way we can keep type same across multiple compilation units, using functors.

```

struct my_lambda {
    int value;
    int operator()(int i) {
        return i + value;
    }
};

auto make_lambda(int value) -> my_lambda {
    return my_lambda{value};
}

using lambda_vector = std::vector<my_lambda>;

auto make_lambda_vector() -> lambda_vector {
    lambda_vector vec;
    vec.push_back(make_lambda(1));
    vec.push_back(make_lambda(1));
    return vec;
}

int main()
{
    auto data = make_lambda_vector();

    data.push_back(make_lambda(1));
    data.push_back(make_lambda(2));
}

```

163. Clang's `-ftime-trace`, this flag is used to get dump file for compile time analysis for building various files, this is for improving build time.

164. Ep 339 - ``static constexpr`` vs ``inline constexpr``

Use of `static constexpr` in header file and including that header file in different cpp file will duplicate the constexpr data, to save duplication we should avoid using `"static constexpr"` and use `"inline constexpr"`. Use `"static constexpr"` at function level and inline at global level. `"Static inline constexpr"` will result in static

165. Ep 343 - Digging Into Type Erasure

Type erasure meaning, hiding the types at runtime

Ex. inheritance, in base class pointer we don't know what derived type of object is coming unless we dynamic cast it.

Another example is the function pointer, we don't know whether it is pointing to lambda, or functor or actual function.

166. Ep 349 - C++23's move_only_function

We have `std::function` which copy the function object but what if we pass the lambda taking `unique_ptr` ? it will not be copied as `unique_ptr` is only moveable, hence to solve this c++23 has introduced

`std::move_only_function()`, it will move the function object . in older c++ we have to use `std::move()` around lambda to work.

```
1  #include <functional>
2  #include <memory>
3
4  void register_callback(std::move_only_function<int(int)> callback);
5
6  int main() {
7      register_callback([
8          // please don't do this
9          [i = std::make_unique<int>(42)](const int val) { return val + *i;
10
11          // what state was cb in?
12      ])
```

167.

Runtime Analysis During Testing

- Address Sanitizer
- Undefined Behavior Sanitizer
- Memory Sanitizer
- Thread Sanitizer
- DrMemory
- valgrind
- Debug Checked Iterators

As Much Static Analysis As Possible

- *at least* `-Wall -Wextra -Wshadow -Wconversion -Wpedantic -Werror` (GCC/Clang) `/W4 /WX`
- `gcc -fanalyzer`
- `cl.exe /analyze`
- `cppcheck`
- `clang-tidy`
- PVS Studio
- Sonar
- *Your favorite (comment!)*

Ship your binary with below option

Control flow is msvc

Fortify source is from gcc

Stack protector is from compiler option

Ubsan minimizes the attack area

Never ship with address sanitiser as it have more attack surface.

Ship With Hardening Enabled

- Control Flow Guard (always deploy with it)
- `_FORTIFY_SOURCE`
- Stack Protector
- UBSan (choose options wisely)
- ~~Address Sanitizer~~ (widens your attack surface area)

168. Lambda object creation

```
Auto l = [](){} ; // this line does not create any object
```

```
l(); // but this line will create an object of lambda type l
```

169. Always run fuz testing on your project, as google and other opensource projects are continuously running fuz testing. If you don't run fuz testing then someone will do and cause a vulneribiltiies attack.

170. Implicit conversion examples

the iterator type of map is different than mentioned one

```
void use_data(const std::map<std::string, std::string> &map) {  
    for (const std::pair<std::string, std::string> &item : map) {  
        std::cout << item.first << ' ' << item.second << '\n';  
    }  
}
```

Use auto& item to avoid copy

```
const char *load_file(const std::filesystem::path &);  
  
std::string_view do_work(std::string_view sv) {  
    const std::string results(load_file(sv));  
    return results.substr(10, 100); string is returned as string view  
}  
  
int main() {  
    const std::string some_path("/home/jason/somefile");  
    do_work(some_path);  
}
```

Implicit conversion happening from string to string view

```
void do_things(const std::shared_ptr<const int> &);  
std::vector<std::shared_ptr<int>> get_data();  
  
void do_yet_more_work() {  
    for (const auto &item : get_data()) {  
        do_things(item); Here, new type of shared pointer is created  
    }  
}
```

New shared pointer is of const int type is created instead of int type, it lead to serious bug

171. With #embed "filename" input(5)

It will dump 5 chars from file as it is in the cpp file

172. Basic_string::resize_and_overwrite vs basic_string::resize

Basic_string::resize_and_overwrite resize the container and overwrite the value given whereas, basic_string::resize does the same but have to provide value only, basic_string::resize_and_overwrite can take lambda also.

173. Union can be anonymous, at a time only member will be active, size of union is size of largest size of a member, there is no default member activated,

```
// unions can be anonymous (unnamed)
// only one member is active (alive) at a time
// you can only access the currently active member
// * any other access IS undefined behavior
// No member is the default member
// Unions can have constructors and destructors in C++11!
// * but destructors are almost impossible to get correct!
// unions support regular member functions
// we need manual bookkeeping to know the active member
// there is a paper for compile-time querying active member
// * https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2641r1.html
// use std::optional or std::variant if that's what you want!
```

174. std::valarray is the class for representing and manipulating arrays of values. It supports element-wise mathematical operations and various forms of generalized subscript operators, slicing and indirect access.

175. Manually calling destructor on local variable is undefined behaviour.

A constructor does not exist

Constructors are the Conversion operators, we can call

Struct s { S(int)}

As

(S)42

This will call a constructor with one parameter.

176. CPM For Trivially Easy Dependency Management With CMake
For cmake package manager

177. Ep 375 - Using IPO and LTO to Catch UB, ODR, and ABI Issues

178. Ultimate CMake C++ Starter Template

Ready templates are available for cmake project on json turner github link

https://github.com/cpp-best-practices/cmake_template

179. Should You Ever std::move An std::array

Ans is yes if array of object having heap allocation or pointer, array of stack based object has no meaning of moving ex. Array of int vs array of string. String object can be moved as it has dynamically allocated pointer.

180. Ep 379 - clang-tidy's "Easily Swappable Parameters" Warning - And How to Fix It!


```

enum Min : int;
enum Max : int;

auto clamp(int value, Min min, Max max) -> int
{
    // return std::max(min, std::min(max, value));
}

int main(const int argc, const char *[])
{
    return clamp(1, Min{100}, Max{argc});
}

```

If all three parameter are of int then it could introduce the bug by overlooking, so introduce a type that enforce to look at definition of function

181. Ep 382 - The Static Initialization Order Fiasco and C++20's `constinit`
 Example final code <https://compiler-explorer.com/z/reb6h91bb>

```

// global, non-const variable
// that is "constant initialized"
// which means, initialized strongly
// before any dynamic code is executed
// (that is, at compile-time)
//
// this is just validating my assumptions
// since it had a constexpr constructor
// it would have been constant initialized
// anyhow
constinit extern Provider global_provider;

// 0) Strongly avoid global mutable
//    objects
//
// 1) global statics that rely on each

```

```
// other are bad, because we cannot
// guarantee initialization order
// (Static Initialization Order Fiasco)
//
// 2) constexpr constructible statics
// are naturally "constant initialized"
//
// 3) constexpr does not change the meaning
// of our code, it validates our
// assumptions
```

182. Ep 386 - C++23's Lambda Attributes

It allow all function attributes on lambda

183. Ep 390 - constexpr + mutable

It not possible to have lambda as constexpr and mutable both,

If we want to modify the data in constexpr lambda then we have to make member as mutable.

```
struct Lambda {
    mutable int i = 0;

    constexpr auto operator()() const {
        return ++i;
    }
};

int main()
{
    constexpr Lambda l1{};

    return l1();
}
```

184. Std::views::enumerate

For sequential container, index : value

For map, key : value

For tuple, value1 : value2 : value3 :


```
constexpr static auto v = {'A', 'B', 'C', 'D'};

for (auto const [index, letter] : std::views::enumerate(v))
    std::cout << '(' << index << ':' << letter << ") ";
std::cout << '\n';

// create a map using the position of each element as key
auto m = v | std::views::enumerate | std::ranges::to<std::map>();

for (auto const [key, value] : m)
    std::cout << '[' << key << "]: " << value << ' ';
```

185. Ep 392 - Google's Bloaty McBloatface

This is software provided by google and it is **used to get the stats of binary size**

It is used to compare two binaries for where the size has grown to

It will give stats about which portion of code taking what size etc.

<https://github.com/google/bloaty>

Its profiler

\$./bloaty bloaty -d compileunits

FILE SIZE	VM SIZE	
34.8%	10.2Mi	43.4% 2.91Mi [163 Others]
17.2%	5.08Mi	4.3% 295Ki third_party/protobuf/src/google/protobuf/descriptor.cc
7.3%	2.14Mi	2.6% 179Ki third_party/protobuf/src/google/protobuf/descriptor.pb.cc
4.6%	1.36Mi	1.1% 78.4Ki third_party/protobuf/src/google/protobuf/text_format.cc
3.7%	1.10Mi	4.5% 311Ki third_party/capstone/arch/ARM/ARMDisassembler.c
1.3%	399Ki	15.9% 1.07Mi third_party/capstone/arch/M68K/M68KDisassembler.c
3.2%	980Ki	1.1% 75.3Ki
third_party/protobuf/src/google/protobuf/generated_message_reflection.cc		
3.2%	965Ki	0.6% 40.7Ki third_party/protobuf/src/google/protobuf/descriptor_database.cc
2.8%	854Ki	12.0% 819Ki third_party/capstone/arch/X86/X86Mapping.c
2.8%	846Ki	1.0% 66.4Ki third_party/protobuf/src/google/protobuf/extension_set.cc
2.7%	800Ki	0.6% 41.2Ki
third_party/protobuf/src/google/protobuf/generated_message_util.cc		
2.3%	709Ki	0.7% 50.7Ki third_party/protobuf/src/google/protobuf/wire_format.cc
2.1%	637Ki	1.7% 117Ki third_party/demangle/third_party/libcxxabi/cxa_demangle.cpp
1.8%	549Ki	1.7% 114Ki src/bloaty.cc
1.7%	503Ki	0.7% 48.1Ki third_party/protobuf/src/google/protobuf/repeated_field.cc
1.6%	469Ki	6.2% 427Ki third_party/capstone/arch/X86/X86DisassemblerDecoder.c
1.4%	434Ki	0.2% 15.9Ki third_party/protobuf/src/google/protobuf/message.cc
1.4%	422Ki	0.3% 23.4Ki third_party/re2/re2/dfa.cc
1.3%	407Ki	0.4% 24.9Ki third_party/re2/re2/regexp.cc
1.3%	407Ki	0.4% 29.9Ki third_party/protobuf/src/google/protobuf/map_field.cc
1.3%	397Ki	0.4% 24.8Ki third_party/re2/re2/re2.cc

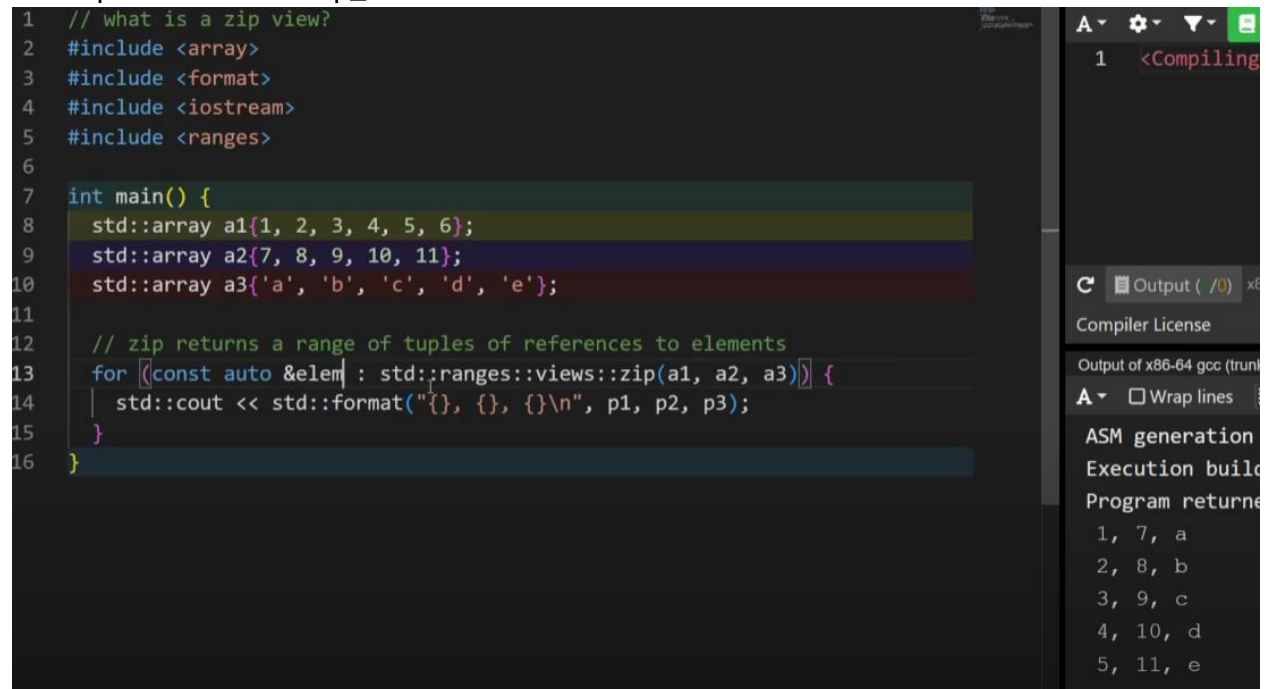
100.0% 29.5Mi 100.0% 6.69Mi TOTAL

186. Delegating Constructors

constructor can call another constructor from same or base class.

187. C++ 20 has new chrono lib think before using boost for time zone, now c++ standard has itself.

188. Ep 398 - C++23's zip_view



```
1 // what is a zip view?
2 #include <array>
3 #include <format>
4 #include <iostream>
5 #include <ranges>
6
7 int main() {
8     std::array a1{1, 2, 3, 4, 5, 6};
9     std::array a2{7, 8, 9, 10, 11};
10    std::array a3{'a', 'b', 'c', 'd', 'e'};
11
12    // zip returns a range of tuples of references to elements
13    for (const auto &elem : std::ranges::views::zip(a1, a2, a3)) {
14        std::cout << std::format("{} {}, {}\n", p1, p2, p3);
15    }
16 }
```

Output (/0) x86_64

Compiler License

Output of x86-64 gcc (trunk)

ASM generation

Execution build

Program returns

```
1, 7, a
2, 8, b
3, 9, c
4, 10, d
5, 11, e
```

189. Ep 399 - C++23's slide_view vs adjacent_view

Adjacent, 2 is for how many elements adjacent

```
int main() {
    std::array a1{1, 2, 3, 4, 5, 6};

    // adjacent returns a range of tuples of references to elements
    // in a sliding window
    for (const auto &[p1, p2] : std::ranges::views::adjacent<2>(a1)) {
        std::cout << std::format("[{}, {}]\n", p1, p2);
    }
}
```

2
3
4
5
6
7
8

Output

filtered

Output of x86-64

A ▾ □ Wrap

ASM generation

Execution

Program n

[1, 2]
[2, 3]
[3, 4]
[4, 5]
[5, 6]

```
// adjacent returns a range of tuples of references to elements
// in a sliding window
// useful when the window size IS known at compile time
for (const auto &[p1, p2] : std::ranges::views::adjacent<2>(a1)) {
    std::cout << std::format("[{}, {}]\n", p1, p2);
}

// slide returns a range of ranges of references to elements
// in a sliding window
// useful when the window size is not known at compile time
for (const auto &window : std::ranges::views::slide(a1, 3)) {
    for (const auto &elem : window) {
        std::cout << elem << ", ";
    }
    std::cout << '\n';
}
```

190. C++23's chunk view and stride view

191. Dogbolt: The Decompiler Explorer

192. <https://undo.io/>

Time travel debugging for c++ application

How to Double (or Triple) Your Salary (and some interview tips)

Networking! (The Human Kind) (effectively "brand building")

1. Get involved in your community

- conferences
- local user groups
- twitter
- reddit
- podcasts
- open source

2. Discover what you are passionate about

- performance
- safety
- the latest features
- best practices

3. Find an opportunity to talk about the things you are passionate about

- meetup/conference lightning talks
- meetup/conference full talks

4. Be explicit about your goals

- Say "I am looking for a job"
- Put it on your slides
- End your talk with a slide that says you are looking for a job

5. Congratulations! Your Job Search (should be/is) Much Easier!

Interviewing

Note: I haven't interviewed in a very long time.

(but I *have* conducted about 300 interviews for podcasts)

- Be aware of Dunning-Kruger effect (and remember it goes both directions)
- Don't undersell or oversell, just be honest
- Don't be afraid to not know the answer (same for speaking)
- Avoid or be careful with the terms "expert" or "guru"
- Demonstrate curiosity
- Ask them questions (same goes for speaking!)
 - Makes you more memorable

- Helps you understand the company culture better
- What questions should I ask?
 - What is your testing culture?
 - What is the CI setup?
 - What are the training budget and learning opportunities?
 - Can I keep speaking/contributing to the community?
- Practice speaking about the things you are passionate about.

We all feel like imposters

Know how much you want to earn

194. Ep 412 - Possible Uses of C++23's [[assume]] Attribute

Give hint to compiler about your code

```

int get_value(int *ptr)
{
    if (ptr == nullptr) {
        throw "Ooops";
    }
    return *ptr;
}

// I promise to never return nullptr
// (why not use a reference then?!?!)
int *get_ptr();

int main()
{
    int *ptr = get_ptr();
    [[assume(ptr != nullptr)]];
    return get_value(ptr);
}

```

```

16      mov     rdi, rax
17      call    __cxa_throw
18  main:
19      sub     rsp, 8
20      call    get_ptr()
21      mov     eax, DWORD PTR [rax]
22      add     rsp, 8
23      ret

```

Output of x86-64 gcc (trunk) (Compiler #1)

Compiler returned: 0

It has completely eliminated call to get_value we can see in asm language code

195. Ep 413 - (2x Faster!) What are Unity Builds

Make unity build flag as true

196. Ep 414 - C++26's Placeholder Variables With No Name

Variable with no name can be declared with `_`, there can be multiple variable with same name
Use case could be to unused structural binding name

```

std::tuple<int, double, float> get_values();

int main()
{
    const auto &[count, __, __] = get_values();

    return count;
}

```

197. Ep 417 - Turbocharge Your Build With Mold
Faster linking tool for linux

198. Important features of c++23

https://github.com/lefticus/cpp_weekly/issues/229

```

/// #1 <print>

#include <print>

/// #2 import std;
/// not really usable yet, compilers :(

/// #3 stack trace

#include <stacktrace>

void my_func() {
    std::println("{} ", std::stacktrace::current());
}

int main(int argc, const char *[])
{
    std::println("{} arguments passed to main! Hello World", argc);
    my_func();
}

```

```

}

/// #4 flat_map and flat_set

// new container adapters for contiguously
// allocated map and set behavior
// by default uses std::vector

// not constexpr capable :(

/// #5 Multidimensional subscript

struct my_md_type {
    int value;
    int &operator[](int , int , int ) { return value; }
};

void use_md(my_md_type &md)
{
    md[2,3,5] = 42;
}

/// #6 Ranges Upgrades!

// zip - easy iteration over multiple containers at once!
// plus a whole lot more!

/// #7 constexpr `cmath`

// common math functions now available at compile time!
// except for trig ones :( (those come in C++26)

```

```

/// #8 <expected>
#include <expected>

std::expected<std::string, int> get_value(bool input) {
    if (input) {
        // non error case
        return "Hello world!";
    } else {
        // or returning an error!
        return std::unexpected(42);
    }
}

void use_expected() {
    auto result = get_value(true);
    // result is an expected object which is an owning container
    // of a string or an int
}

/// #9 <generator>

// first coroutine support!!!

/// #10 <mdspan>

// #include <mdspan>

//void use_md(std::mdspan<6,7,8> md)
//{
//    md[2,3,5] = 42;
//}

```



```

/// #11 explicit `this`
struct S {
    void go(this auto &&self) {
        // write one version for both const and non-const!
    }
};

/// #12 size literal suffixes

void sz_things() {
    auto value = 42z; // signed size_t (ptrdiff_t)
    auto value2 = 42uz; // size_t
}

/// #13 start_lifetime_as

// no support so I cannot demonstrate it!

// allows us to avoid reinterpret_cast by
// telling the compiler "there's an object of that type in
// that memory location!!"

```

199. Return the exact object and not try to create by implicit conversion

```

std::optional<Lifetime> get_value()
{
    return Lifetime{42};
}

```

Above code will eliminate RVO, it will create 2 object, one is with default construction and another with move construction.

To gain RVO return exact object.

Return `std::optional<Lifetime>{42};`

200. Attribute summery

```

// C++11
// [[noreturn]]
// [[carries_dependency]] for memory ordering

// C++14
// [[deprecated("optional reason")]]

// C++17
// [[fallthrough]]
// [[nodiscard]]
// [[maybe_unused]]

// C++20
// [[likely]]
// [[unlikely]]
// [[no_unique_address]] multiple member can have same address

// C++23
// [[assume]]

```

201. C++23 has first support for std::generator

202. C++ Weekly - Ep 429 - C++26's Parameter Pack Indexing

```

template <typename... Param>
void func26(Param... param) {
    // do something with 3rd param
    param...[2] = 42;
}

```

Directly modifying index of parameter pack

203. String implementation using small object optimisation

<https://compiler-explorer.com/z/a3q6x3Eas>

204. Ep 434 - GCC's Amazing NEW (2024) -Wnrvo

205. Ep 435 - Easy GPU Programming With AdaptiveCpp (68x Faster!)

206. Lambda capture

```
template<typename ... String>
void work_with_string_like_things(const String & ... string)
{
    auto l = [&]() {
        ( (std::cout << string << '\n'), ...);
    };

    l();
}

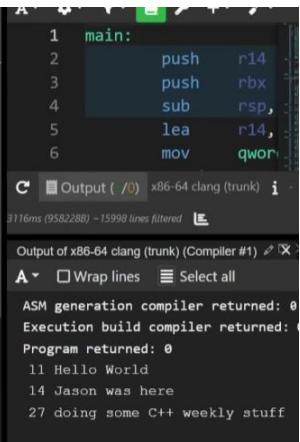
int main()
{
    work_with_string_like_things("Hello World", std::string{"Jason was here"},
        std::string_view{"doing some C++ weekly stuff"});
}
```

It support parameter packs for lambda

```
template<typename ... String>
void work_with_string_like_things(const String & ... string)
{
    auto l = [...strlike = std::string_view(string)]() {
        ( (std::cout << strlike.size() << ' ' << strlike << '\n'), ...);
    };

    l();
}

int main()
{
    work_with_string_like_things("Hello World", std::string{"Jason was here"},
        std::string_view{"doing some C++ weekly stuff"});
}
```



1 main:
2 push r14
3 push rbx
4 sub rsp,
5 lea r14,
6 mov qword
Output of x86-64 clang (trunk) (Compiler #1)
ASM generation compiler returned: 0
Execution build compiler returned:
Program returned: 0
11 Hello World
14 Jason was here
27 doing some C++ weekly stuff

207. Pointers to lambda

```
void use_callable(auto){}
```

```

void callable(int);
void callable(double);

void callable_template(auto);

int main()
{
    //Note we can convert the function pointer to the appropriate
    //overload by static casting to the desired signature.
    use_callable(static_cast<void (*)(double)>(&callable));
    use_callable(static_cast<void (*)(char)>(&callable_template));
    // not really necessary for int, but if we want to think
    // of this generically
    use_callable([]<typename Param>(Param &&v) -> decltype(auto)
    {
        if callable has multiple overload then this template will convert it to appropriate function pointer
        return callable(std::forward<Param>(v));
    });
}

```

208. 438 - C++23's ranged-for Fixes

C++23 extend the lifetime of temp object in ranged for loop

Before c++23, the lifetime get ended as soon as the expression evalauted and causing undefined behaviour

Ex. for(const auto& x : Object()) {}

209. Ep 439 - mutable (And Why To Avoid It)

210. C++17's variadic using

```

template<typename ... Callable>
struct visitor : Callable... {
    using Callable::operator()...;
};

```

```

std::variant<int, float, std::string_view> get_variant()
{
    return 4.2f;
}

template<typename ... Callable>
struct visitor : Callable... {
    using Callable::operator()...;
};

int main()
{
    const auto value = get_variant();
    std::visit(
        visitor{
            [](int i){fmt::print("Int: {}\n", i);},
            [](float f){fmt::print("Float: {}\n", f);},
            [](std::string_view sv){fmt::print("SV: {}\n", sv);},
        }, value);
}

```

211. Ep 441 - What is Multiple Dispatch

```

#include <fmt/format.h>

#include <string_view>
#include <variant>

struct SpaceObject {
    constexpr virtual ~SpaceObject() = default;
    [[nodiscard]] virtual constexpr auto get_name() const
noexcept
    -> std::string_view = 0;
    int x;
}

```

```

    int y;
};

struct Craft : SpaceObject {
    [[nodiscard]] constexpr std::string_view get_name()
const noexcept override {
    return "Craft";
}
};

struct Asteroid : SpaceObject {
    [[nodiscard]] constexpr std::string_view get_name()
const noexcept override {
    return "Asteroid";
}
};

template <typename... Callable>
struct visitor : Callable... {
    using Callable::operator()...;
};

void collide(const Craft &, const Craft &) {
    std::puts("C/C"); }
void collide(const Asteroid &, const Asteroid &) {
    std::puts("A/A"); }
void collide(const Craft &, const Asteroid &) {
    std::puts("C/A"); }

```

```

void collide(const Asteroid &, const Craft &) {
    std::puts("A/C"); }

std::vector<std::variant<Craft, Asteroid>> get_objects();

void process_collisions(const std::variant<Craft,
Asteroid> &obj) {
    for (const auto &other : get_objects()) {
        std::visit(
            [](const auto &lhs, const auto &rhs) {
                if (lhs.x == rhs.x && lhs.y == rhs.y) {
                    collide(lhs, rhs);
                }
            },
            obj, other);
    }
}

```

Here we are getting multiple object in variant and in for loop we have used lambda, where it has auto parsed it between lhs and rhs. And we have defined 4 variants of collide method, using this we can achieve the multi method dispatch.

212. Ep 444 - GCC's Implicit constexpr

Use -fimplicit-constexpr flag that will make a function automatic inline a that satisfy the const expression requirements

Ex. inline auto make_value() { return 43; }

213. C++23 std::forward_like

```

auto get_value(this auto &&self) -> auto && {
    return std::forward_like<decltype(self)>(self.value);
}

```

It is used to get return type as template parameter.

Ex. it is expecting type of self.value same as self

214. Declaring destructor disables the move constructor.

215.