Changes in standards C++11 and C++14

Andrzej Grzenda

23-04-2020

Introduction

- **C++ evolution**
  - **Bjarne Stroustrup works since 1979**
  - **ANSI C++ Committee founded 1990**
  - **Official standards: C++98, C++03, C++11, C++14, C++17, C++20**
- **Reference sources**
  - **cppreference.com**
  - **isocpp.org**
  - **www.open-std.org/JTC1/SC22/WG21/**

C++ reference

C++98, C++03, C++11, C++14, C++17, C++20

A LOT was changed in C++11

- `auto`
- `decltype`
- trailing func return
- Lambda expressions
- Move semantics
- Smart pointers
- New std containers
- Tuple
- Thread support
- New algorithms
- Aliases with `using`
- Range-based `for` loop
- `nullptr`
- Uniform initialization
- `initializer_list`
- `constexpr`
- `default`
- `delete`
- `override`
- `final`
- Non-static members init
- Delegating constructors
- Explicit conversions
- Parameter packs
- Variadic templates
- Enum enhancements
- attributes
- `static_assert`
- New libraries
- Data alignment
- `noexpect`
- Variable template
- Type traits
- New numeric types
- User-defined literals
- New string literals

`auto` keyword – type deduction

`auto` - placeholder type specifier

`auto` was a very rare keyword before C++11.

Now, the point is to deduce the type from the initializer:

```
                  int var1 = 5;
                 auto var2 = 5;
         const auto param1 = 5;
          const int param2 = 5;
            int& paramRef1 = param;
           auto& paramRef2 = param;
vector<int>::iterator pos1 = findElem(vec, 4);
                 auto pos2 = findElem(vec, 4);
```

`auto` - placeholder type specifier

Examples:

```
//What is the result???
auto result1 = runCalculations(some_parameter);

//Oh, it’s a vector:
std::vector<int> result2 = runCalculations(some_parameter);
```

`auto` - placeholder type specifier

Examples:

```
std::vector<int>::iterator it1 = std::find(myVec.begin(), myVec.end(), valueToFind);
                      auto it2 = std::find(myVec.begin(), myVec.end(), valueToFind);
```

```
for(auto it = myVec.begin(); it != myVec.end(); ++it){
        processElem(*it);
}
```

`auto` return type – C++14

Compiler can deduce the return type from the return statements.
All of them must deduce to the same type:

```
auto myFunc(bool param){
       if (param) return 0.5f;       //float
       else return 1.5;              //double => compilation error!
}
```

`decltype` keyword

`decltype` - placeholder type specifier

The point is to obtain a type identical to another type:

`decltype(`*expression*`)`

- **returns type of *expression***
- **DOES NOT EXECUTE the *expression***
- **includes const, volatile, &**

Example: create a vector of a map’s keys

```
auto map1 = someDataContainer.getMap();    //std::map of something to something
vector<decltype(map1)::key_type> keys1;    //a vector of keys
```

`decltype(auto)` – C++14

Simplifies syntax:

```
decltype(someCalculations()) calcResult = someCalculations();
                decltype(auto) calcResult = someCalculations();
```

Changes deduction rules:

```
void someFun(const int& input){
        auto var1 = input;              //int
        decltype(auto) var2 = input;    //const int&
}
```

trailing function return type

`auto functionName() -> returnType;`

- **Syntax could be more readable.**
- **Return type is visible, although `auto`.**

Use case: return type depends on parameters

```
auto myFunc(double input) -> double{
        if (input <= 0) return 1;
        else return 1/input;
}
```

Conflicting types! So can’t use auto return type.

lambda expressions

Lambdas are anonymous, local functions

```
auto closureType = [captures](parameters){body};
```

Captures:

- **Allow access to variables outside lambda body**
- **[&] is a default capture by reference**
- **[=] is a default capture by copy**
- **Any variable can be explicitly captured**

Lambda example: sort algorithm with comparison function

```
vector<string> toSort = {"zz","aaa","d","4444"};
auto compFun = [](const auto& str1, const auto& str2){
               return str1.length()<str2.length();
               };

sort(toSort.begin(), toSort.end()); //{"4444","aaa","d","zz"};


sort(toSort.begin(), toSort.end(), compFun);//{"d","zz","aaa","4444"};
```

`auto` parameter type:
a generic lambda
(since C++14)

move semantics

Move semantics: new object classification

Intuitevely:

- lvalue has an identifiable address
- rvalue has no identifiable address

```
int var1 = 4;
int var2 = var1 + 2;
int& ref1 = var1;
//int&& ref2 = var2; //error!
void myFunc(int& input){}     //chosen for var1, var2, ref1
void myFunc(int&& input){}    //chosen for "var1+2"
```

Motivation: to avoid copying

```
class LargeStruct {
        LargeStruct(const LargeStruct& other){//copy};
        LargeStruct operator=(const LargeStruct& other){//copy};
        LargeStruct(const LargeStruct&& other){//move};
        LargeStruct operator=(const LargeStruct&& other){//move};
};


vector<LargeStruct> container(5);
container[0] = LargeStruct();       //default constructor and move
container[1] = buildLargeStruct();  //arbitrary build function and move
```

smart pointers

shared_ptr

```
string pointerName = new string("text");

shared_ptr<string> pointerName = shared_ptr<string>(new string("text"));
```

- **manages any object**
- **can be copied to multiple owners**
- **has operator*, operator->**

Like an ordinary pointer

- **<u>object is automatically deleted when all owners are out of scope</u>**

shared_ptr

two allocations!

```
auto ptr = shared_ptr<string>(new string("text"));
{
        auto p1 = make_shared<string>("text");
        //p1.use_count==1
        auto p2 = p1;
        //p1.use_count==2; p2.use_count==2
        auto p3 = move(p1);
        //p1.use_count==0; p2.use_count==2; p3.use_count==2
}
//variables p1, p2, p3 go out of scope
//memory is freed
```

one allocation

unique_ptr

- **manages any object**
- **deletes the owned object when going out of scope**
- **can have only one owner**
- **can not be copied**
- **has operator*, operator->**

unique_ptr

```
//usual usage:
unique_ptr<string> data1 = unique_ptr<string>(new string("text"));
              auto data2 = unique_ptr<string>(new string("text"));

//shortest version:
auto data3 = make_unique<string>("text");            //since C++14

//uniqueness:
//data2 = data1;              //error!
  data2 = move(data1);        //correct

//convertible to bool, has operator* and operator->
if(data1) cout<<*data1<<" "<<data1->size();
```

weak_ptr

- a weak reference does not own an object

- must be converted to shared_ptr in order to access it

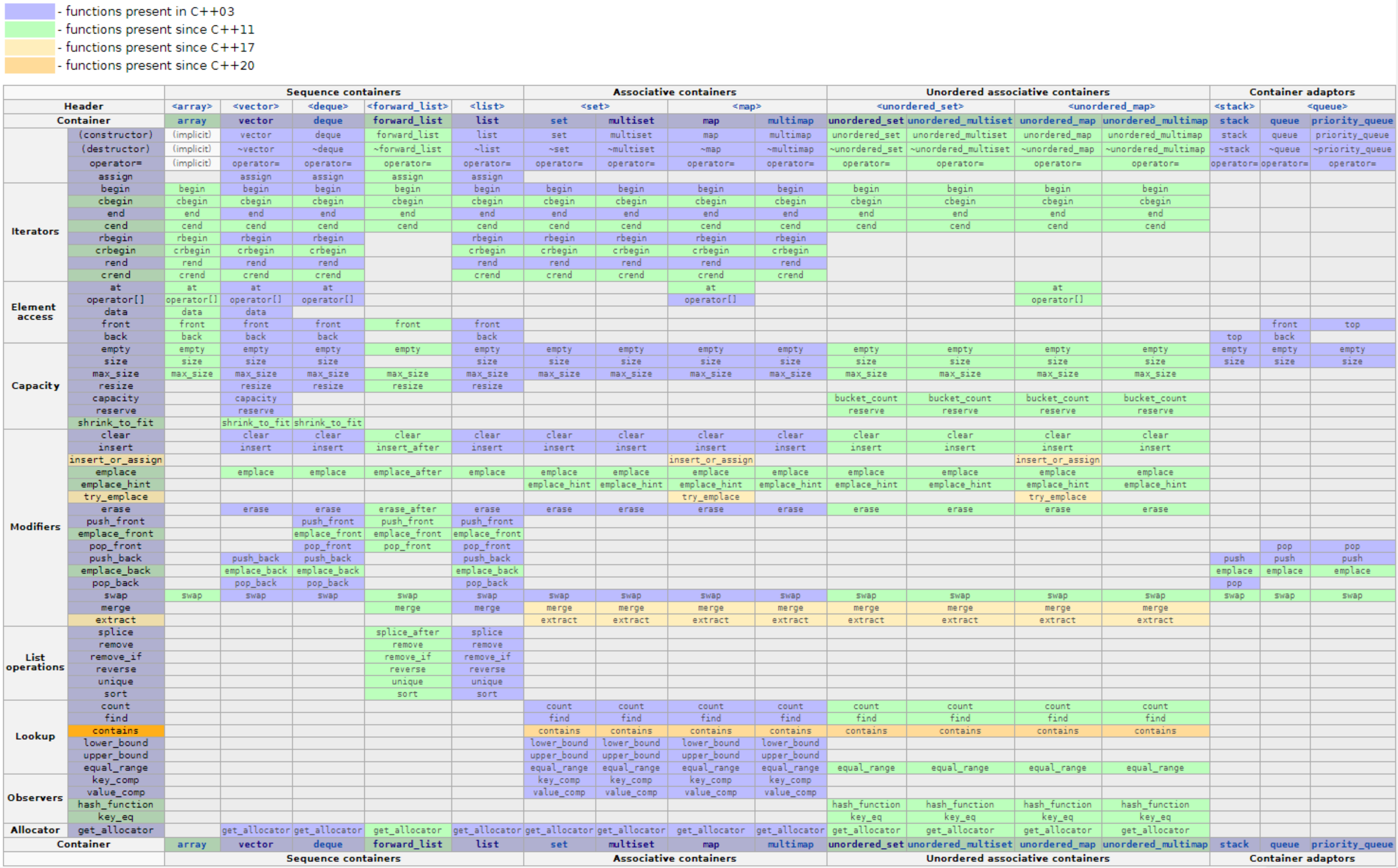
  consequence: no does not have operator*, operator->

- can be used to avoid circular references

weak_ptr

```
struct Object{
        shared_ptr<string> ownedResource;
        weak_ptr<string> notOwnedResource;
};
int main(){
        Object obj;
        {
                auto sp1 = make_shared<string>("1st resource");
                auto sp2 = make_shared<string>("2nd resource");
                obj.ownedResource = sp1;
                obj.notOwnedResource = sp2;
        }//sp1, sp2 out of scope
        //obj.ownedResource "1st resource" exists
        //obj.notOwnedResource "2nd resource" is deleted
}
```

containers enhancements



🟦 - functions present in C++03
🟩 - functions present since C++11
🟨 - functions present since C++17
🟧 - functions present since C++20

		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		
Header		<array>	<vector>	<deque>	<forward_list>	<list>	<set>		<map>		<unordered_set>		<unordered_map>		<stack>	<queue>	
Container		array	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(constructor)	(implicit)	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(destructor)	(implicit)	~vector	~deque	~forward_list	~list	~set	~multiset	~map	~multimap	~unordered_set	~unordered_multiset	~unordered_map	~unordered_multimap	~stack	~queue	~priority_queue
	operator=	(implicit)	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=
	assign		assign	assign	assign	assign											
Iterators	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin			
	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin			
	end	end	end	end	end	end	end	end	end	end	end	end	end	end			
	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend			
	rbegin	rbegin	rbegin	rbegin		rbegin	rbegin	rbegin	rbegin	rbegin							
	crbegin	crbegin	crbegin	crbegin		crbegin	crbegin	crbegin	crbegin	crbegin							
	rend	rend	rend	rend		rend	rend	rend	rend	rend							
	crend	crend	crend	crend		crend	crend	crend	crend	crend							
Element access	at	at	at	at					at				at				
	operator[]	operator[]	operator[]	operator[]					operator[]				operator[]				
	data	data	data														
	front	front	front	front	front	front										front	top
	back	back	back	back		back									top	back	
Capacity	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty
	size	size	size	size		size	size	size	size	size	size	size	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size			
	resize		resize	resize	resize	resize											
	capacity		capacity								bucket_count	bucket_count	bucket_count	bucket_count			
	reserve		reserve								reserve	reserve	reserve	reserve			
	shrink_to_fit		shrink_to_fit	shrink_to_fit													
Modifiers	clear		clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear			
	insert		insert	insert	insert_after	insert	insert	insert	insert	insert	insert	insert	insert	insert			
	insert_or_assign								insert_or_assign				insert_or_assign				
	emplace		emplace	emplace	emplace_after	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace			
	emplace_hint						emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint			
	try_emplace								try_emplace				try_emplace				
	erase		erase	erase	erase_after	erase	erase	erase	erase	erase	erase	erase	erase	erase			
	push_front			push_front	push_front	push_front											
	emplace_front			emplace_front	emplace_front	emplace_front											
	pop_front			pop_front	pop_front	pop_front										pop	pop
	push_back		push_back	push_back		push_back									push	push	push
	emplace_back		emplace_back	emplace_back		emplace_back									emplace	emplace	emplace
	pop_back		pop_back	pop_back		pop_back									pop		
	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap
	merge				merge	merge	merge	merge	merge	merge	merge	merge	merge	merge			
	extract						extract	extract	extract	extract	extract	extract	extract	extract			
List operations	splice				splice_after	splice											
	remove				remove	remove											
	remove_if				remove_if	remove_if											
	reverse				reverse	reverse											
	unique				unique	unique											
	sort				sort	sort											
Lookup	count						count	count	count	count	count	count	count	count			
	find						find	find	find	find	find	find	find	find			
	contains						contains	contains	contains	contains	contains	contains	contains	contains			
	lower_bound						lower_bound	lower_bound	lower_bound	lower_bound							
	upper_bound						upper_bound	upper_bound	upper_bound	upper_bound							
	equal_range						equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range			
Observers	key_comp						key_comp	key_comp	key_comp	key_comp							
	value_comp						value_comp	value_comp	value_comp	value_comp							
	hash_function										hash_function	hash_function	hash_function	hash_function			
	key_eq										key_eq	key_eq	key_eq	key_eq			
Allocator	get_allocator		get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator			
Container		array	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		

New containers

Fixed-size array: `array<int, 5> arr;`

- **wraps a C-style array**
- **has standard methods: `size()`, `begin()`, `swap()` etc.**

One-way list: `forward_list<int>`

- **introduces a singly-linked list besides the doubly-linked one**
- **is more space efficient**

Unordered versions of old containers:

  - `unordered_set`
  - `unordered_multiset`
  - `unordered_map`
  - `unordered_multimap`

- **faster search, insertion, removal: constant time instead of logarythmic**

New methods in many existing containers

- **Constant iterators assure read-only access:**

```
cbegin()
cend()
crbegin()
crend()
```

- **Constructing an element in-place:**

```
vector<int> myVec;
myVec.emplace_back(3);
myVec.emplace_front(1);
myVec.emplace(myVec.begin()+1, 2);
//{1,2,3}
```

std::tuple

A generalization of `std::pair`

- **Creating:**

```
auto myTuple = make_tuple(1, 1.0, "hi")
```

- **Element access:**

```
int data1 = get<0>(t1);
int data2 = get<int>(t1);
```

- **Tying to variables:**

```
int data1;
double data2;
string data3;
tie(data1, data2, data3) = getDataFromBase(2);
```

Returns a matching tuple

thread support

new algorithms operating on ranges

New functions in `<algorithm>` library

- **Returning bool:**

```
all_of(firstElem, lastElem, predicate)
any_of(firstElem, lastElem, predicate)
none_of(firstElem, lastElem, predicate)
```

- **Returning iterator one past last element:**

```
copy_if(firstElem, lastElem, predicate)
copy_n(firstElem, count, firstOutputElem)
move(firstElem, lastElem, firstOutputElem)
```

`using` keyword - aliases

using instead of typedef

```
typedef int i32;

typedef vector<int*> tVecPtrInt;
```

```
using i32 = int;

using uVecPtrInt = vector<int*>;

template<typename T>
using pT = T*;
```

range-based `for` loop

Range-based `for` loop

Iterating over a collection, like `std::list`:

```
list<int> myList(10);
for (list<int>::iterator it = myList.begin(); it != myList.end(); ++it){
        cout<<*it;
}
for(auto it : myList){
        cout<<it;
}
```

Works for:

- **C-style arrays**
- **Containers from std libraries**
- **User-defined containers meeting specific criteria**

nullptr

`nullptr`

- a null pointer with its own type: `nullptr_t`
- solves problems with type of `NULL`
- any pointer is implicitly convertible to `nullptr`
- imposed by standard, not implementation-defined
- recomended instead of `NULL` or 0

nullptr

```
myFunc(int* input){//implementation};
myFunc(int input){//implementation};
myFunc(NULL);  //ambiguous!
myFunc(0);
myFunc(nullptr);
```

uniform initialization

`std::initializer_list`

Uniform initialization

Harmonisation of writing initializations with {} :

```
int var{2};
int arr[]{1,2,3};
struct twoIntegers{
        int one, two;
};
twoIntegers myFunc(){
        return {1,2};
}
```

initializer_list

A way to initialize types with {}:

```
struct ListInitializable{
        ListInitializable(initializer_list<int> initList){
                /*implementation*/
        }
};
```

A way to pass a {} list to a function:

```
void processSomeInts(initializer_list<int> initList){
        for(auto it: initList){/*processing*/}
};
ListInitializable listInit{5,6,7};
processSomeInts({1,2,3});
```

initializer_list

Caution:

- `vector<int> vec1(5);//{0,0,0,0,0}`
- `vector<int> vec2{5};//{5}`

Because vector has both constructors with `initializer_list` and a single argument:

- `vector(size_type count);`
- `vector(initializer_list<T> init);`

`constexpr` keyword

`constexpr`

- **indicates that the variable or function can be used in constant expressions**

- **this allows compiler to evaluate them in compile-time**

- **important part of C++ philosophy:**

move the computations from execution to compilation whenever possible

- **in C++11, C++14, C++17, C++20 many functions from standard libraries were gradually made `constexpr`**

new keywords:

`default` `override`

`delete` `final`

New `default` meaning

User-defined ctor prevents compiler from generating other ctors, but it can be forced :

```
class myClass{
        myClass(int in);
        myClass() = default;
}
```

Only special member functions can be defaulted:

- **Default constructor**
- **Move and copy constructor**
- **Move and copy operator=**
- **destructor**

New `delete` meaning

Any function can be deleted, so that it can not be used.

Typical use: deleting special member functions:

```
class myClass{
        myClass& operator=(myClass& other) = delete;
        myClass&(myClass& other) = delete;
}
```

Result: objects of myClass can not be copied.

New keyword: `override`

Explicit declaration of overriding a method:

```
struct Base{
        virtual void myFunc1();
        virtual void myFunc2(int x);
};
struct Derived : Base{
        void myFunc1() override;        //overriding
        void myFunc2(float x) override;//mismatch – no overriding
};
```

New keyword: `final`

Indicates that a class can not be inherited from,

or that a method can not be overridden.

```
class Base{
        virtual void myFunc();
};
class Derived final : Base{
        void myFunc() override final;
};
class thisIsWrong : Derived{
        void myFunc() override;
};
```

Delegating construcotrs

Non-static member initialization

Delegating constructor

A constructor can call another constructor in its initializer list:

```
struct MyClass(){
        MyClass(int in1, in2):
                var1(in1), var2(in2)
        {}
        MyClass() : MyClass(5,8)
        {}
        int var1, var2;
};
```

Non-static member initialization

A class member can be initialized directly in declaration:

```
struct MyClass(){
        MyClass(){};
        MyClass(int in) : var1(in){};
        int var1 = 5;
};
MyClass obj1();        //var1==5
MyClass obj2(2);       //var1==2
```

But initialization in constructor initializer list is more important.

`explicit` conversion

Explicit conversion

A way to prevent implicit conversions:

```
struct myStruct {
    operator int(){ return 42; }
    explicit operator int*(){ return nullptr; }
};
myStruct x;
int n = static_cast<int>(x);         //explicit - OK
int m = x;                           //implicit - OK
int* p = static_cast<int*>(x);       //explicit - OK
int* q = x;                          //implicit - error!
```

variadic templates
parameter packs

Variadic templates and parameter packs

Variadic templates accept arbitrary number of arguments.
Typical usage involves recursion:

```
template <typename Head, Typename... Tail>
struct Count<Head, Tail...>{
        static int value = 1+Count<Tail...>::value;
};
template<> struct Count<> {
        static int value = 0;
};

Count<int, double, string>::value;
```

`enum` enhancemets

`enum class` and underlying type

```
enum class Color{red,green,blue};
enum class Hue : unsigned char {pink, red, orange};
```

Underlying type

```
Color firstColor = Color::red;
Color secondColor = red;
```

attributes

attributes

Additional hints for the programmer or the compiler.

Examples:

- **`[[noreturn]]` – function does not return**
- **`[[deprecated]]` – the function should not be used anymore**
- **More in C++17 or implementation-defined**

`static_assert`

static_assert

Compile-time assertion:

```
static_assert(expression, message)
```

If *expression* is false, compile error apperas.

- **prevents bugs**
- **can output clear messages on what’s wrong**

new std libraries

New libraries

`<chrono>` - date and time utilities

`<random>` - classes to generate random and pseudo-random numbers

`<regex>` - regular expressions library

type traits

type_traits

A library for extensive type chacking and manipulation. Examples:

- `is_integral`
- `is_reference`
- `is_final`
- `is_same`
- `is_convertible`
- `remove_const`
- `common_type`

user-defined literals

```
minutes operator""min(int mins);
```

`noexcept` keyword

data alignment:

`alignas`

`alignof`

NOKIA