Changes in standard C++17

Andrzej Grzenda

30-04-2020

Introduction

- **Modern C++ evolution**
  - **Every 3 years since 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

What was changed in C++17

`std::optional`
`std::any`
`std::variant`
Structured bindings
Fold expressions
Mandatory copy elision
`<filesystem>` library
`if` with initializer
`switch` with initializer
`constexpr if`
New algorithms

Execution policies
New standard attributes
Hex floating point literals

Removed:

- Trigraphs
- Dynamic exceptions specification
- auto_ptr
- register keyword

Removed old features

no one wants them anyway

What was removed in C++17?

Trigraphs:

??<	??>	??(	??)	??=	??/	??'	??!	??-
{	}	[	]	#	\	^	|	~

This was used on systems with limited character support.

Sometimes could cause subtle bugs:

```
cout << "Enter birth in format: ??/??/??";   //not what it seems…
cout << "Enter birth in format: \??";
```


NOKIA

What was removed in C++17?

- **`auto_ptr` – useless „smart” pointer**

Because `unique_ptr` and `shared_ptr` are superior in every way.

- **`register` keyword**

Indicated automatic storage and a hinted to put a variable in processor’s register.

In practice, meaningless.

- **`operator++` for `bool` type**

Prior to C++17 booleans could be incremented, which is counter-intuitive

What was removed in C++17?

Dynamic exception specification:

```
void myFun1() throw(exceptionType1, exceptionType2, …);
```

Practically, either:

- **not used – anything can be thrown**
- **`throw()` – nothing can be thrown**

Deprecated in C++11. Another mechanism was introduced:

```
void myFun2() noexcept;
void myFun3() noexcept(constBooleanExpression);
```

`std::optional`

object that may not be there

std::optional

```
optional<int> thisMayBeEmpty;
if(condition)
        thisMayBeEmpty = 5;
…
if(thisMayBeEmpty &&
   thisMayBeEmpty < 10)}
    int value1 = *thisMayBeEmpty;
}
…
thisMayBeEmpty = {};
int value2=thisMayBeEmpty.value();
```

declaration

underlying type → optional

optional → bool

compare contents

dereference an optional (unsafe)

empty contents

throws bad_optional_access

std::optional

Use case: returning from a function.

```
optional<double> myDiv(double num, double denom){
        if (denom != 0.0)
                return num/denom;
        return {};
}
```


`std::any`

like `void*`, but better

std::any

Can store anything safely.

```
any anyVal = 5;
cout << any_cast<int>(anyVal) << "\n";

anyVal = string("aaa");
cout << any_cast<string>(anyVal) << "\n";

anyVal = (int[3]){1,2,3};
cout << any_cast<double>(anyVal) << "\n";    //throws bad_any_cast
cout << any_cast<double>(&anyVal) << "\n";   //returns nullptr
```

`std::variant`

like `union`, but better

std::variant

Can store an object of certain types safely.

```
variant<int, string, float> var1;
var1 = "aaa";
var1 = 5.0f;
//var1 = 'a';                          //throws bad_variant_access

float contentF = get<float>(var1);
      contentF = get<2>(var1);         //identical
//get<int>(var1);                      //throws bad_variant_access
get_if<int>(&var1);                    //returns nullptr
```

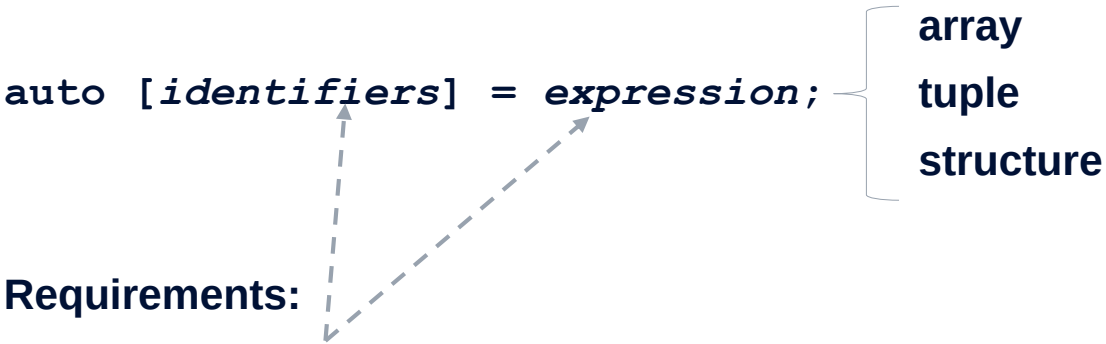

NOKIA

Structured bindings

declare & assign many variables at once!

Structured bindings

A new type of declaration to simplify syntax:



Requirements:

- **both sides must have same number of elements**
- **this number must be known at compile time**

NOKIA

Structured bindings: array

For array

```
int myArr[2] = {42, 3};
auto& [a,b] = myArr;
a = 1;
b = 2;
//myArr contains {1,2}
```

For tuple

```
tuple<int, float> myTuple (42, 3.14);
auto& [a,b] = myTuple;
a = 1;
b = 2;
//myTuple contains {1,2}
```

For struct

```
struct MyStruct{
    int elem1 = 42;
    float elem2 = 3.14;
} myStruct;
auto& [a,b] = myStruct;
a = 1;
b = 2;
//myStruct contains {1,2}
```

Structured bindings: use case

```
double b;
std::tie(ignore,b) = someTuple;        //since C++11, only for tuples


//C++17
auto [c,d] = someTuple;
auto [e,f] = someArray;
auto [g,h] = someStruct;
```


Fold expressions

```
sum = (variables + … + 8);
```

Fold expressions

When a range needs to be consolidated with an operator, use std::accumulate:

```
accumulate(myVec1.begin(), myVec1.end(),
        1,
        [](auto a, auto b){return a*b;}
        );
```

Can it be done in a function for any numer of parameters?

```
multiplySomething(2.0, 3.5, 4.2, 5.55);
```

Existing methods:

```
template<typename T>
double multiplySth1(initializer_list<T> list){
         double result = 1.0;
         for (auto elem: list) result*=elem;
         return result;
}
multiplySth1({2.0, 3.5, 4.2, 5.55});
```

```
double multiplySth2(){
        return 1.0;
}
template<typename T, typename ... Args>
double multiplySth2(T input, Args... args){
        return input * multiplySth2(args...);
}
multiplySth2(2.0, 3.5, 4.2, 5.55);
```


NOKIA

Fold expression:

```
template<typename... Args>
double multiplySth3(Args... args){
        return (args * ...);
}
multiplySth3(2.0, 3.5, 4.2, 5.55);
```

Fold to right:

```
(pack operator ...)
(pack operator ... operator init)
```

Fold to left:

```
(... operator pack)
(init operator ... operator pack)
```

Assignment example:

```
template<typename... Args>
void multiAssign(int value, Args&... args){
        (args = ... = value);
}
int a,b,c;
multiAssign(5,a,b,c);                  // exactly like a=b=c=5;
```

Fold expression for empty list

Empty pack is OK for operators:	`&&`	`||`	`,`
Returned values are:	`true`	`false`	`void()`

```
template<typename... Args>
double andElements(Args... args){
        return (args && ...);
}
andElements(true, false, true);       // -> false
andElements(true, true);              // -> true
andElements();                        // -> true
```

Mandatory copy elision

Elision of move and copy operations

```
someObject foo(){
        return someObject();
}
void bar(someObject obj){…}


auto var1 = foo();     //no copy/move constructor
bar(someObject());     //no copy/move constructor
```

Works even when copy/move constructors are explicitly deleted.

Exception specification is now a part of type

Exception specification is a part of type

Two almost identical declarations:

```
void noexceptFun1() noexcept(true);
void noexceptFun2() noexcept(false);
```

Demonstration:

```
is_same<decltype(noexceptFun1), decltype(noexceptFun2)>::value;
//true before C++17, false since C++17
```

Filesystem library

NOKIA

Filesystem library

A bunch of useful functions, supporting:

- **Paths**
- **File properties**
- **File type**
- **Directory properties**
- **Copying**
- **Renaming**
- **And more…**

`if` and `switch` with initializer

`if` with initializer

A handy syntax enhancement:

```
set<int> mySet{1,2,3,4,5};
auto it = mySet.find(6);
if(it!=mySet.end())
        cout<<"Found!\n";
else
        cout<<"not found\n";
```

```
if(auto it = mySet.find(6); it!=mySet.end())
```

`switch` with initializer

A handy syntax enhancement:

```
set<int> mySet{1,2,3,4,5};
auto it = mySet.find(3);
switch(*it){
        case 1:cout<<"first\n"; break;
        case 5:cout<<"last\n"; break;
        default: cout<<"middle\n"; break;
}
```

```
switch(auto it = mySet.find(3); *it)
```

constexpr if

NOKIA

constexpr if

The statement of `constexpr if` is discarded, if the condition is false.

```
unsigned int input;
if constexpr (is_floating_point<decltype(input)>::value)
        cout<<"Float!\n";
if constexpr (is_signed<decltype(input)>::value)
        cout<<"Signed!\n";
```

new algorithms

NOKIA

Some additions to algorithm library

- `for_each_n`

Applies a function to n elements of a sequence.

- `sample`

Picks n random elements from a sequence, with a given RNG.

- `clamp`

Accepts a value and 2 limits (lower, upper) and returns one of them.

- `reduce`

Similar to `accumulate`, but not in default order. Has execution policy.

Execution policies

NOKIA

Execution policies

Many algorithms from `std` can be executed in parallel:

- `find`
- `all_of`
- `for_each`
- `count`
- `search`
- `copy`
- `fill`
- `move`
- `transform`
- `generate`
- `remove`
- `replace`
- `sort`
- `is_sorted`

Execution policies

Algoritms are executed according to one of policies from `std::execution`:

- **Sequenced: `seq`**

Indicates that algorithm can not be parallelized.

- **Parallel: `par`**

Algorithm can be parallelized. Execution in a single thread is sequenced.

- **Unsequenced: `unseq`**

Algorithm can be vectorized: an operation is performed once on multiple data.

- **Parallel unsequenced: `par_unseq`**

Algorithm can be parallelized or vectorized. Execution in a single thread is unordered.

new attributes

New attributes

`[[fallthrough]]` – used in switch statement to indicate that a fallthrough is OK:

```
switch(myVar){
        case 0:
                fun0();         //warning might be here
        case 1:
                fun1();
        case 2:                  //no warning
                fun2();
                [[fallthrough]];
        default: break;
}
```

New attributes

`[[nodiscard]]` – for functions which return value should not be ignored

```
[[nodiscard]] bool checkTheResult(){……}
checkTheResult();       //warning
if (checkTheResult())  //correct
{……}
```

If applied to `class` or `struct`, means: if this is ever returned by a function, don't ignore it!

NOKIA

New attributes

`[[maybe_unused]]` – suppresses warnings about unused entities:

- **class**
- **typedef**
- **data member**
- **variable**
- **function**
- **enumerator**

Some attributes enhancements

- **There is a numer of non-standard attributes; if any of them is not recognized, it is ignored by compiler without error.**

- **Attributes can be in namespaces, so keyword `using` was enabled:**

```
[[ attribute1, attribute2, attribute3 ]]
[[ space1::attr1, space2::attr2 ]]
[[ using space1: attr1, attr2, attr3 ]]
```

Modern initialization of enums

Direct-list-initialization of enumerations

Conditions:

- **only for enum class or explicit underlying type:**

```
enum class myColorSet : char{ red, green, blue };
```

- **only for one-element initialization list:**

```
myColorSet myCol1 = myColorSet::red;
myColorSet myCol2 {myColorSet::red};
myColorSet myCol3 {0};
myColorSet myCol4 = 0;
```

- **any conversion can't be narrowing:**

```
myColorSet myCol5 {999};
```


NOKIA

Range-based for loop: different types of `begin()` and `end()`

NOKIA

Range-based for loop: different types of range `begining()` and `end()`

```
struct dummy{
        int arr[5] = {0,1,2,3,4};
        int*  begin(){ return arr; }
        void* end() { return arr+5; }
};

for(auto elem : dummy()){
        cout<<elem<<"\n";
}
```


hex floating literals

```
0xA0.4p3
```

Hex floating literals

```
double var1 = 12.2e-1;
```

$$\left(1 \times 10^{1} + 2 \times 10^{0} + 2 \times 10^{-1}\right) \times 10^{-1} = 1,22$$

```
double var2 = 0xA0.4p3
```

$$\left(10 \times 16^{1} + 0 \times 16^{0} + 4 \times 16^{-1}\right) \times 2^{3} = 1282$$

NOKIA