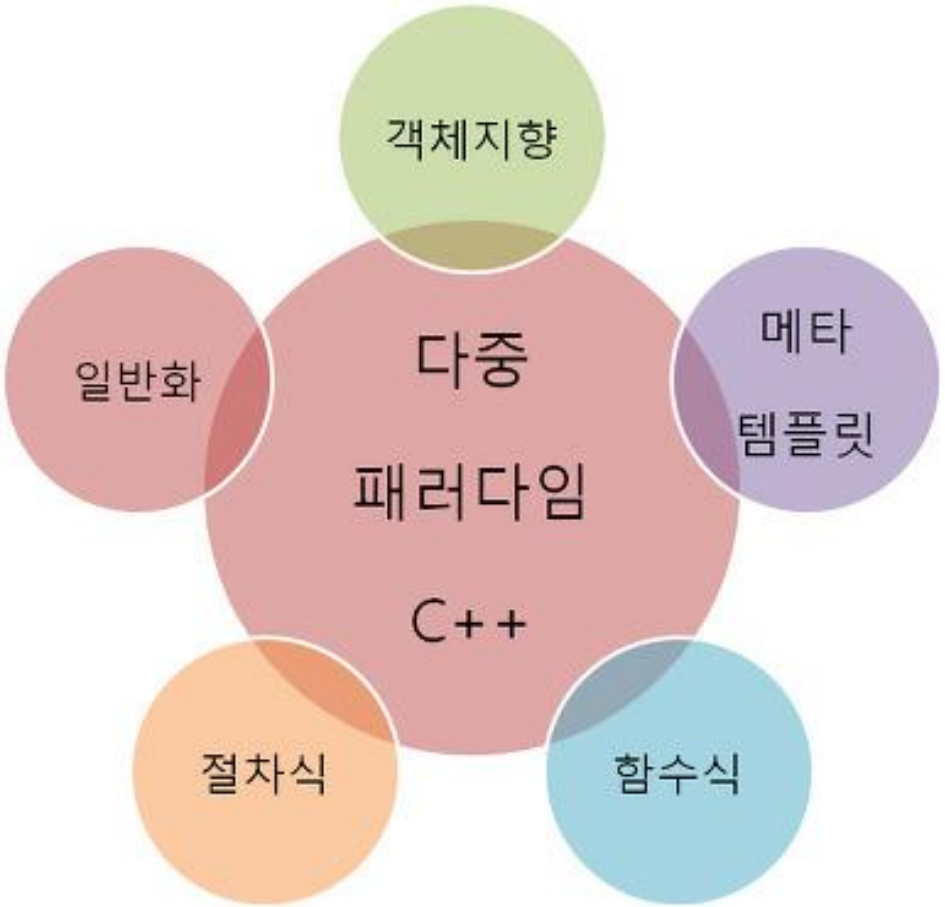
VK_Prerequisite

Generic programming and syntactic sugar in modern C++

Contents

- Generic programming?
- Lambda function & Function object
- Type deduction
- and a little bit more…

Generic programming?



Lambda function

- A function definition that is not bound to an identifier

Syntax

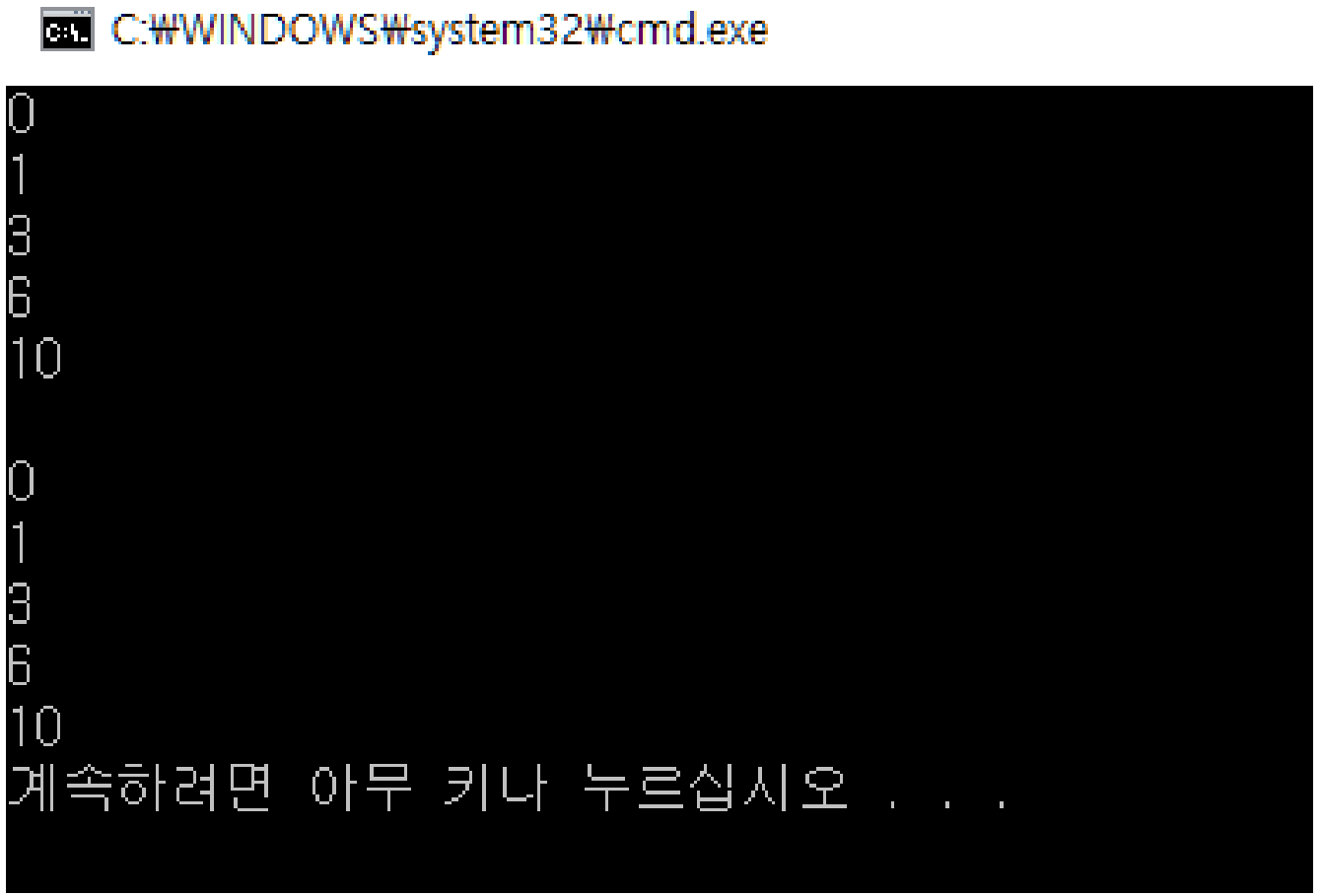
`[` *capture-list* `]` `(` *params* `)` **`mutable`**(optional) **`constexpr`**(optional)(c++17) *exception attribute* `->` *ret* `{` *body* `}`	(1)
`[` *capture-list* `]` `(` *params* `)` `->` *ret* `{` *body* `}`	(2)
`[` *capture-list* `]` `(` *params* `)` `{` *body* `}`	(3)
`[` *capture-list* `]` `{` *body* `}`	(4)

Function Object vs Lambda

```
44
45  class Sum
46  {
47  public:
48
49      int a;
50
51      Sum() : a(0) {}
52
53      int operator()(int b)
54      {
55          a += b;
56
57          return a;
58      }
59  };
60
```

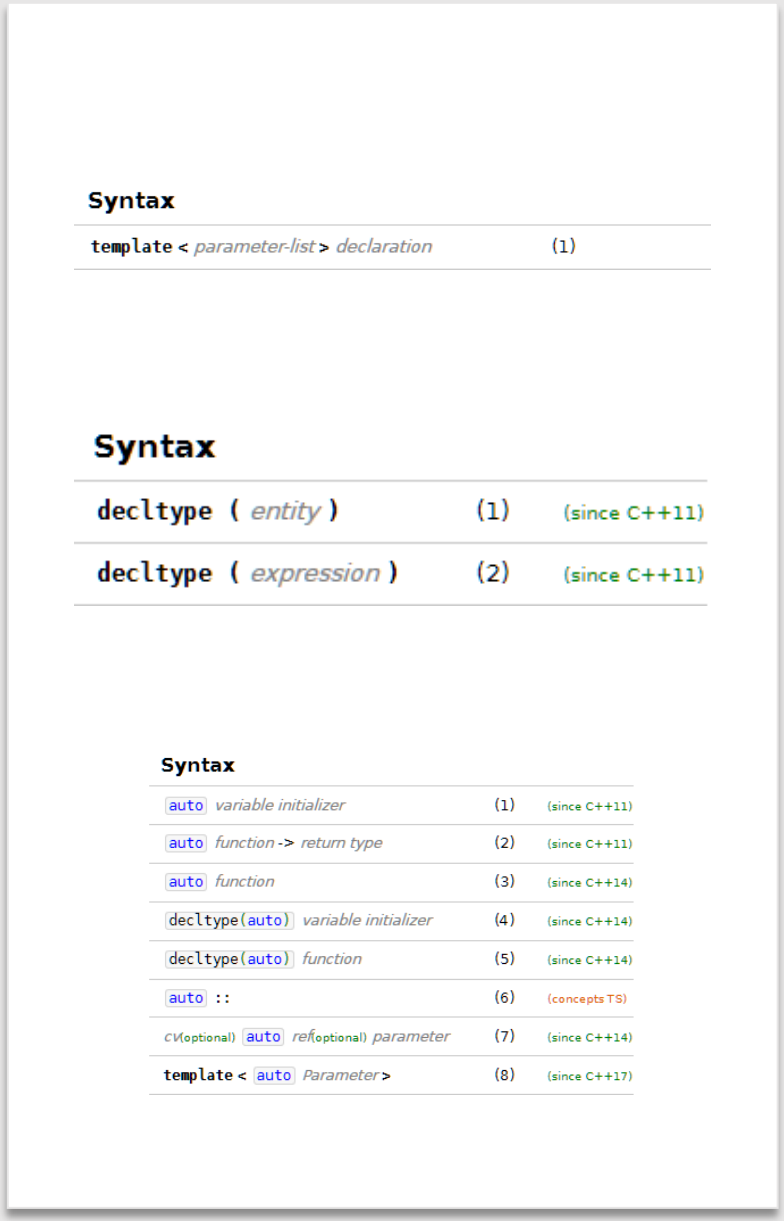
```
60
61  int main()
62  {
63      std::vector<int> vec;
64
65      for (int i = 0; i < 5; i++) vec.push_back(i);
66
67      Sum sum;
68
69      int total = 0;
70
71      //Lambda
72      auto lambdaSum = [&total](int i)
73      {
74          total += i;
75          std::cout << total << std::endl;
76      };
77
78      //Functor
79      for (int i : vec) std::cout << sum(i) << std::endl;
80
81      std::cout << std::endl;
82
83      for (int i : vec) lambdaSum(i);
84
85      return 0;
86  }
```

Function Object vs Lambda



Pros

- 1. Can ‘capture’ the state of a function
- 2. Lazy evaluation
- 3. Widely used in many powerful libraries(Ex. STL, Boost…)



Syntax

`template` < *parameter-list* > *declaration* (1)

Syntax

`decltype` (*entity*)	(1)	(since C++11)
`decltype` (*expression*)	(2)	(since C++11)

Syntax

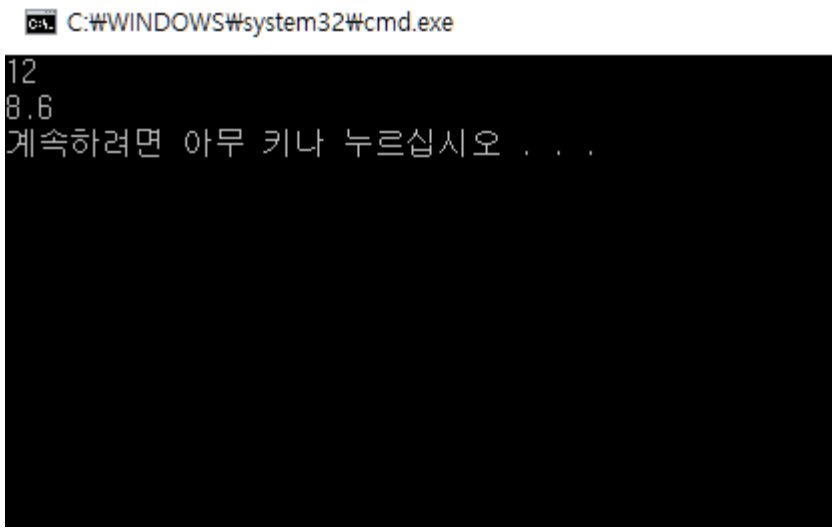
`auto` *variable initializer*	(1)	(since C++11)
`auto` *function* -> *return type*	(2)	(since C++11)
`auto` *function*	(3)	(since C++14)
`decltype(auto)` *variable initializer*	(4)	(since C++14)
`decltype(auto)` *function*	(5)	(since C++14)
`auto` ::	(6)	(concepts TS)
cv(optional) `auto` *ref*(optional) *parameter*	(7)	(since C++14)
`template` < `auto` *Parameter* >	(8)	(since C++17)

Type deduction

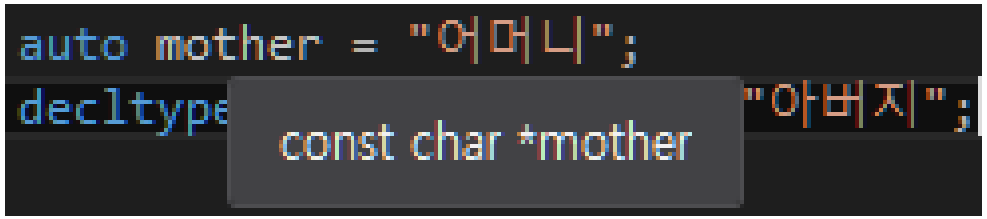
- template
- decltype
- auto

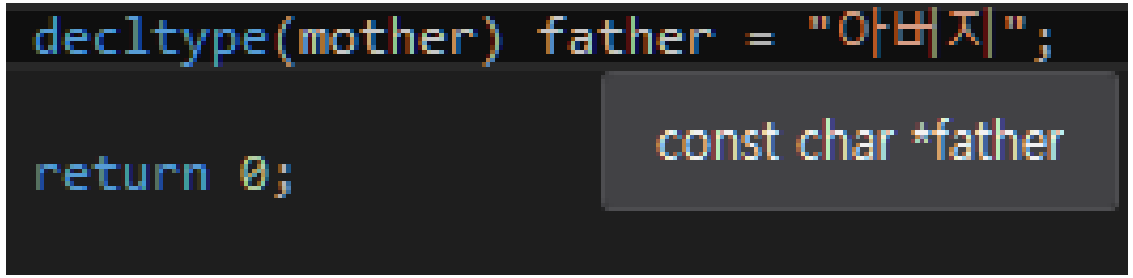
Template

```
71
72  template <typename T>
73  class Sum
74  {
75  public:
76
77      T operator()(T a, T b)
78      {
79          return a + b;
80      }
81  };
82
83  int main()
84  {
85      Sum<int> sumint;
86      Sum<double> sumdouble;
87
88      std::cout << sumint(5, 7) << std::endl;
89      std::cout << sumdouble(1.3, 7.3) << std::endl;
90
91      return 0;
92  }
```



auto & decltype





Example

```
6
7   int main()
8   {
9       // Get random number
10
11      std::random_device rd;
12      std::default_random_engine gen(rd());
13      std::uniform_int_distribution<int> dist(1, 9);
14      std::vector<int> vec1;
15
16      auto roll = std::bind(dist, gen);
17
18      for (int i = 0; i < 5; i++) vec1.push_back(roll());
19
20      std::function<void(int)> PrintVec = [](int i) // std::function
21      {
22          std::cout << i << " ";
23      };
24
25      //Multiplying all elements
26
27      int multiple = 1;
28
29      auto Multi = [&multiple](int i) // auto
30      {
31          multiple *= i;
32      };
33
34      //Execution
35
36      std::for_each(vec1.begin(), vec1.end(), PrintVec); // for_each
37      std::cout << std::endl;
38      for (int i : vec1) Multi(i); // Range-based for loop
39
40      std::cout << multiple << std::endl;
41
42      return 0;
43  }
```

Example