Placeholder Syntax

Let's learn about placeholder syntax in this lesson.

With `auto`, C++11 has unconstrained placeholders. We can use concepts in C++20 as constrained placeholders. Decisive quantum leap does not look so thrilling at the first glimpse. C++ templates will become easy to use C++ features.

According to our definition, C++98 is not a consistent language. By consistent, we mean that you have to apply a few rules to derived C++ syntax from it. C++11 is something in between. For example, we have consistent rules like initializing all with curly braces (see { } - Initialization). Of course, even C++14 has a lot of features where we miss a consistent principle. One of the favorites is the generalized lambda function.

Placeholder Syntax: `auto`

```
auto genLambdaFunction= [](auto a, auto b) {
   return a < b;
};

template <typename T, typename T2>     // 3
auto genFunction(T a, T2 b){           // 4
   return a < b;
}
```

Inconsistency in C++14

By using the placeholder `auto` for the parameter `a` and `b`, the generalized lambda function becomes - in a magic way - a function template. We know, `genLambdaFunction` is a function object that has an overloaded call operator which accepts two type parameters. The `genFunction` is also a function template. But wouldn't it be nice to define a function template by just using `auto` in a function definition? This would be consistent but is not possible. Hence, we have to use a lot more syntax (line 3 and 4). That syntax is often too difficult for a lot of C++ programmer.

Exactly that inconsistency will be removed with the placeholder syntax. Therefore, we have a new simple principle and C++ will become - according to my definition - a lot easier to use.

> Generic Lambdas introduced a new way to define templates.

Constrained and Unconstrained Placeholders

We will get unconstrained and constrained placeholders. `auto` is an unconstrained placeholder because `a` with `auto` defined variable can be of any type. A `concept` is a constrained placeholder because it can only be used to define a variable that satisfies the `concept`.

> **General Rule:** Constrained Concepts can be used where unconstrained templates (`auto`) are usable.

Let's define and use a simple `concept` before we dig into the details.

Example:

```
// conceptsPlaceholder.cpp

#include <iostream>
#include <type_traits>
#include <vector>

template<typename T>
concept bool Integral(){
  return std::is_integral<T>::value;
```

```
}

Integral getIntegral(auto val){

  return val;
}

int main(){

  std::cout << std::boolalpha << std::endl;

  std::vector<int> myVec{1, 2, 3, 4, 5};
  for (Integral& i: myVec) std::cout << i << " ";
  std::cout << std::endl;

  Integral b= true;
  std::cout << b << std::endl;

  Integral integ= getIntegral(10);
  std::cout << integ << std::endl;

  auto integ1= getIntegral(10);
  std::cout << integ1 << std::endl;

  std::cout << std::endl;

}
```

We have defined in line 8 the `concept Integral`. The `concept Integral` will evaluate to `true` if the predicate `std::is_integral<T>::value` returns `true` for `T`. `std::is_integral<T>` is a function of the type-traits library. The functions of the type-traits library enable, amongst other things, that we can check types at compile-time. Hence, we iterate over `Integral`'s in the range-based for-loop in line 21 and the variable `b` in line 24 has to be `Integral`. Our usage of concepts goes on in lines 27 and 30. We required in line 27 that the return type of `getIntegral` (line 12) has to fulfill the concept `Integral`. We're not so strict in line 30. Here we're fine with an unconstrained placeholder.

In the next lesson, we'll discuss the predefined concepts in C++20.