- Examples

Let's look at how auto is being used in two different scenarios.

WE'LL COVER THE FOLLOWING

Replacing basic data types

```
#include <iostream>
#include <vector>

int func(int){ return 2011;}

int main(){

  auto i = 5;
  auto& intRef = i;             // int&
  auto* intPoint = &i;         // int*
  const auto constInt = i;     // const int
  static auto staticInt = 10;  // static int

  std::vector<int> myVec;
  auto vec = myVec;           // std::vector<int>
  auto& vecRef = vec;         // std::vector<int>&

  int myData[10];
  auto v1 = myData;           // int*
  auto& v2 = myData;          // int (&)[10]

  auto myFunc = func;          // (int)(*)(int)
  auto& myFuncRef = func;      // (int)(&)(int)

  // define a function pointer
  int (*myAdd1)(int, int) = [](int a, int b){return a + b;};

  // use type inference of the C++11 compiler
  auto myAdd2 = [](int a, int b){return a + b;};

  std::cout << "\n";
  // use the function pointer
  std::cout << "myAdd1(1, 2) = " << myAdd1(1, 2) << std::endl;


  // use the auto variable
  std::cout << "myAdd2(1, 2) = " << myAdd2(1, 2) << std::endl;

  std::cout << "\n";

}
```

Explanation

In the example above, the types are automatically deduced by the compiler, based on the value stored in the variable. The corresponding types of variables are mentioned in the in-line comments.

- In line 8, we have defined a variable, `i`, and its type is deduced to be `int` because of the value `5` stored in it.
- In lines 9-12, we have copied the values into different variables and their type is deduced `auto`-matically based on the value stored in it.
- Similarly, in lines 15-16, we are copying a vector and the reference to it using the assignment operator, `=`. The `auto` keyword takes care of `vec` and `vecRef` types.
- In lines 22-23, `auto` determines the type of `myFunc` to be a function pointer and `myFuncRef` as a reference to the function.
- In line 29, we have defined a lambda expression whose return type is inferred by the C++ compiler since we have used the `auto` keyword.

> We have used lambda functions in lines 26 and 29, click here to study more about them.

Advanced types

```
#include <chrono>
#include <future>
#include <map>
```

```
#include <string>
#include <tuple>


int main(){

  auto myInts = {1, 2, 3};
  auto myIntBegin = myInts.begin();

  std::map<int, std::string> myMap = {{1, std::string("one")}, {2, std::string("two")}};
  auto myMapBegin = myMap.begin();

  auto func = [](const std::string& a){ return a;};

  auto futureLambda= std::async([](const std::string& s ) {return std::string("Hello ") + s;}

  auto begin = std::chrono::system_clock::now();

  auto pa = std::make_pair(1, std::string("second"));

  auto tup = std::make_tuple(std::string("first"), 4, 1.1, true, 'a');
}
```

Explanation

- In this example, we can see how `auto` is used with different libraries and data structures.
- The compiler automatically infers the correct type for the given value.
- This makes `auto` a very useful feature since determining or declaring the types for different libraries can be a cumbersome task.

In the next lesson, there is a coding challenge to test the concepts we've covered.