Automatic Type Deduction: auto

In this lesson, we will discuss the automatic type deduction using auto.

WE'LL COVER THE FOLLOWING

The Facts of `auto`

Automatic type deduction with `auto` is extremely convenient. Firstly, we save unnecessary typing, in particular with challenging template expressions. Secondly, the compiler does not make human errors.

The compiler automatically deduces the type from the initializer:

```
auto myDoub = 3.14;
```

Key Features

- The techniques for automatic function template argument deductions are used.
- It is very helpful in complicated template expressions.

- It empowers us to work with unknown types.
- It must be used with care in combination with initializer lists.

The following code compares the definition of explicit and deduced types:

```
#include <vector>

int myAdd(int a,int b){ return a+b; }

int main(){

  // define an int-value
  int i= 5;                                  // explicit
  auto i1= 5;                                // auto

  // define a reference to an int
  int& b= i;                                 // explicit
  auto& b1= i;                               // auto

  // define a pointer to a function
  int (*add)(int,int)= myAdd;               // explicit
  auto add1= myAdd;                         // auto

  // iterate through a vector
  std::vector<int> vec;
  for (std::vector<int>::iterator it= vec.begin(); it != vec.end(); ++it){}
  for (auto it1= vec.begin(); it1 != vec.end(); ++it1) {}

}
```

> 💡 C++ Insights helps us to visualize of the types that the compiler deduces. Andreas Fertig, author of this tool, wrote a few [blog] enteries (https://www.modernescpp.com/index.php/c-insights-type-deduction) about `auto` as well.

`auto`-matically Initialized

`auto` determines its type from an initializer, meaning that without an initializer, there is no type and nor variable. Simply put, the compiler takes care of each type that is initialized. This is a nice side effect of `auto` that is rarely mentioned.

It makes no difference if we forgot to initialize a variable or did not make it because we failed to understand the language The result is the same: undefined behavior. With `auto` we can overcome these errors undefined behavior. With `auto`, we can overcome these errors.

Moving on from that overview, let's implement auto in a few examples. Before moving on, do you know all the rules for the initialization of a variable? If yes, congratulations! Let's move forward. If not, read the article default initialization and all referenced articles in this article before continuing with the examples.

The aforementioned article states that "objects with automatic storage duration (and their sub-objects) are initialized to indeterminate values". This formulation causes more harm than good. Local variables that are not user-defined will not be initialized by default.

In the following samples, we modified the second program of default initialization to make the undefined behavior clearer.

Sample Code 1

```
// init.cpp

#include <iostream>
#include <string>

struct T1 {};

struct T2{
    int mem;     // Not ok: indeterminate value
 public:
    T2() {}
};

int n;          //  ok: initialized to 0

int main(){

  std::cout << std::endl;

  int n;               // Not ok: indeterminate value
  std::string s;       // ok: Invocation of the default constructor; initialized to ""
  T1 t1;               // ok: Invocation of the default constructor
  T2 t2;               // ok: Invocation of the default constructor

  std::cout << "::n " << ::n << std::endl;
  std::cout << "n: " << n << std::endl;
  std::cout << "s: " << s << std::endl;
  std::cout << "T2().mem: " << T2().mem << std::endl;

  std::cout << std::endl;

}
```

Explanation 1

First, let us discuss the scope resolutions operator `::` is used in line 25. `::` addresses the global scope. In our case, it is the variable `n` in line 14.

Curiously enough, the automatic variable `n` in line 25 has the value 0. `n` has an undefined value; therefore, the program has undefined behavior. This is also true for the variable `mem` of the `struct` `T2` since `T2().mem` returns an undefined value.

Sample Code 2

Now, we will rewrite the program with the help of `auto`.

```
// initAuto.cpp

#include <iostream>
#include <string>

struct T1 {};

struct T2{
    int mem = 0;  // auto mem= 0 is an error
 public:
    T2() {}
};

auto n = 0;

int main(){

  std::cout << std::endl;

  using namespace std::string_literals;

  auto n = 0;
  auto s = ""s;
  auto t1= T1();
  auto t2= T2();

  std::cout << "::n " << ::n << std::endl;
  std::cout << "n: " << n << std::endl;
  std::cout << "s: " << s << std::endl;
  std::cout << "T2().mem: " << T2().mem << std::endl;

  std::cout << std::endl;

}
```

Explanation 2

Two lines in the source code are especially interesting. Firstly, in line 9, the current standard forbids the code to initialize non-constant members of a `struct` with `auto`. Therefore, we must use an explicit type. For more on the C++ standardization committee regarding this issue, read this: article.

Secondly, in line 23, C++14 gets C++ string literals. We build them by using a C string literal ("") and adding the suffix s ("" `s`). For convenience, we already imported that in line 20: `using namespace std::string_literals`.

The output of the program is not as thrilling as it is only for completeness. `T2().mem` has the value 0.

Refactorization

`auto` supports the refactorization of our code. Firstly, it is very easy to restructure our code when there is no type information. Secondly, the compiler automatically takes care of the right types. What does that mean? We will give an answer in the form of a code snippet. Firstly, examine the code without `auto`:

```
int a = 5;
int b = 10;
int sum =  a * b * 3;
int res = sum + 10;
```

When we replace the variable b of type `int` by a `double` 10.5, we must adjust all dependent types, which is laborious and dangerous. We must use the right types to handle the issue of narrowing and other *intelligent phenomenons* in C++.

```
int a2 = 5;
double b2 = 10.5;
double sum2 = a2 * b2 * 3;
double res2 = sum2 * 10.5;
```

Sample Code 3

This danger is not present in case of `auto`. Everything happens `auto`-matically. Let us an example of this:

```
// refactAuto.cpp

#include <typeinfo>
#include <iostream>

int main(){

  std::cout << std::endl;

  auto a = 5;
  auto b = 10;
  auto sum =  a * b * 3;
  auto res = sum + 10;
  std::cout << "typeid(res).name(): " << typeid(res).name() << std::endl;

  auto a2 = 5;
  auto b2 = 10.5;
  auto sum2 = a2 * b2 * 3;
  auto res2 = sum2 * 10;
  std::cout << "typeid(res2).name(): " << typeid(res2).name() << std::endl;

  auto a3 = 5;
  auto b3 = 10;
  auto sum3 = a3 * b3 * 3.1f;
  auto res3 = sum3 * 10;
  std::cout << "typeid(res3).name(): " << typeid(res3).name() << std::endl;

  std::cout << std::endl;

}
```

Explanation 3

The small variations of the code snippet always determine the right type of `res`, `res2`, or `res3` which is the job of the compiler. The variable `b2` in line 17 is of type `double` and therefore, `res2` is also double.

The variable `sum3` in line 24 becomes a `float` due to multiplication with the literal float `3.1f` (a `float` type). Therefore, the final result, `res3`, is also a `float` type. To access the data type from the compiler, we have used the `typeid` operator which is defined in the header `typeinfo`.

> 💡 Often, developers in the embedded domain do not need the correct type but rather a concrete type, such as `int`. In the case of this example, this is a nice trick. When we switch from the implicit type with auto to

> the concrete type within, we must make the assignment with the help of
>
> `{}` braces `(int res3 = {sum3 * 10};)`. Thanks to `{}` the compiler checks if a narrowing conversion has taken place. If we get an error, we know the current type is not what we expected.

Let’s take a look at an example of this topic in the next lesson.