Automatic Type Deduction: auto

In this lesson, we'll learn how C++ can automatically deduce the data type of a variable.

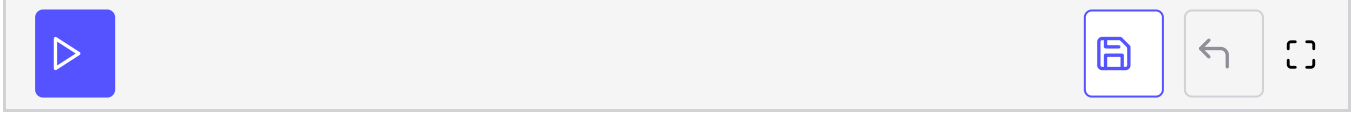
WE'LL COVER THE FOLLOWING

One of the best features shipped in C++11 was automatic type deduction.

The `auto` keyword can be used to let the compiler decide the data type itself.

We no longer need to explicitly define data types, which is a big help, especially when we're dealing with complex template expressions.

```
#include <iostream>

int main() {
  auto myStr = "Educative";
  auto myDoub = 3.14;
  auto myInt = 3;
  std::cout << myStr << std::endl;
  std::cout << myDoub << std::endl;
  std::cout << myInt << std::endl;
}
```



Key features

- The techniques for automatic function template argument deductions are used.

- It's very helpful in complicated template expressions.
- It enables us to work with unknown types:

```
auto func=[]{ return 5; };
```

> We have used a lambda function in the statement above, click here to learn more about it.

It has to be used with care in combination with initializer lists:

```
auto myInt{2011};
auto myInt2= {2011};
```

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 visualize the types that the compiler deduces.

> Andreas Fertig, the author of this tool, wrote a few blog posts about `auto` as well.

`auto`-matically initialized

`auto` determines its type from an initializer. That simply means that; without an initializer, there is no type and therefore, no variable. In simpler terms, the compiler ensures that each type is initialized. This is a nice side effect of `auto` which is rarely mentioned.

It makes no difference whether we forget to initialize a variable or didn't initialize it because of a misunderstanding of the language. The result is the same: **undefined behaviour**. With `auto`, we can overcome these nasty errors.

Do we know all the rules for the variable initialization? If yes, congratulations. If not, read the article default initialization and all referenced articles in the given link.

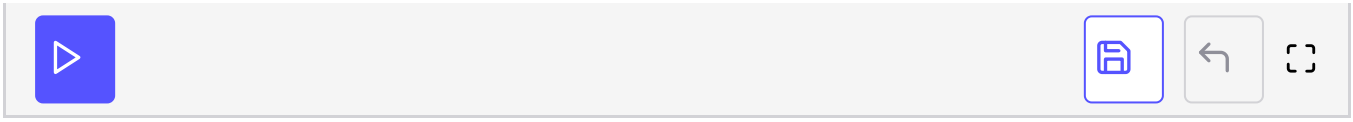
The article contains the following statement: "objects with automatic storage duration (and their sub-objects) are initialized to indeterminate values". This optimization causes more harm than good. Local variables that are not user-defined will not be default initialized.

I modified the second program to default initialization to make the undefined behavior more obvious.

Undefined behavior

```
// 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;

}
```



First, let's talk about the scope resolution operator, `::`, in line 25. `::` addresses the global scope. In our case, it is the variable `n` in line 14.

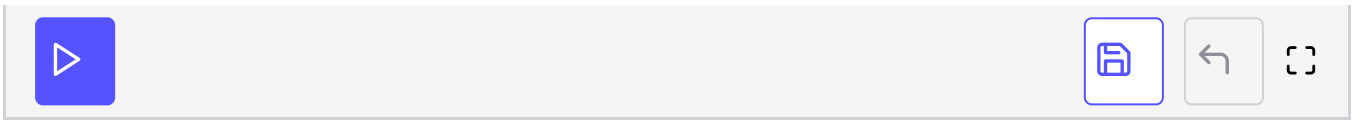
Curiously enough, the automatic variable, `n`, in line 20 has the value `0`. `n` has an undefined value and therefore the program has undefined behavior. This will also hold for the variable `mem` of the class `T2` as `T2.mem` returns an undefined value.

Using `auto`

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;

}
```



Two lines in the source code are especially interesting. Firstly, in line 9, the current standard forbids us to initialize non-constant members of a class with `auto`. Therefore, we have to use an explicit type. This perspective is counterintuitive. Here is a discussion by the C++ standardization committee about this issue: article.

Second, as we can see in line 23, C++14 gets C++ string literals. We build them by using a C string literal, `""`, and add the suffix s, `""s`. For convenience, we imported that in line 20: `using namespace std::string_literals`.

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

Further information

- C++ Insights
- blogs on auto
- default initialization
- C++ standardization committee article

In the next lesson, we'll learn how to refactor code with `auto`.