Uniform Initialization with {}

In this lesson, we will learn about initialization with {} and how it prevents narrowing.

WE'LL COVER THE FOLLOWING

The initialization of variables was uniform in C++11.

Two Forms of Initialization

> **Rule:** A {}-Initialization is always applicable.

Direct Initialization

```
string str{"my String"};
```

Copy Initialisation

```
string str = {"my String"};
```

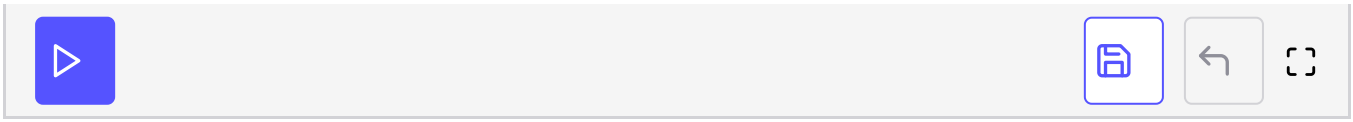
Preventing Narrowing

The initialization with `{}` prohibits narrowing conversion.

Narrowing, or more precisely narrowing conversion is an implicit conversion of arithmetic values. This can lead to a less accurate result which is extremely dangerous.

The following example outlines the issue with the classical initialization for fundamental types. It doesn't matter whether we use direct initialization or assignment.

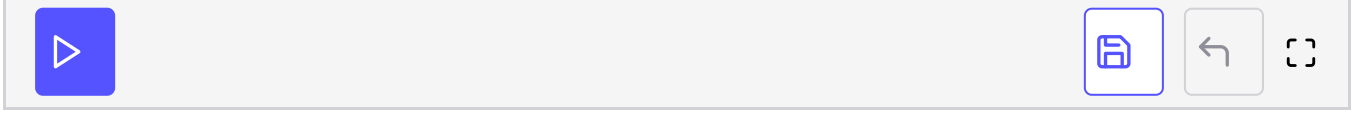
```
#include <iostream>

int main(){

  char c1(999);
  char c2= 999;
  std::cout << "c1: " << c1 << std::endl;
  std::cout << "c2: " << c2 << std::endl;

  int i1(3.14);
  int i2= 3.14;
  std::cout << "i1: " << i1 << std::endl;
  std::cout << "i2: " << i2 << std::endl;

}
```



The output of the program shows that there are two warnings in the code. First, the `int` literal 999 does not fit into the type `char`. Second, the `double` literal does not fit into the `int` type.

That is not possible with {}-initialization.

```
// narrowingSolved.cpp

#include <iostream>

int main(){

  char c1{999};
  char c2 = {999};
  std::cout << "c1: " << c1 << std::endl;
  std::cout << "c2: " << c2 << std::endl;

  int i1{3.14};
  int i2 = {3.14};
  std::cout << "i1: " << i1 << std::endl;
  std::cout << "i2: " << i2 << std::endl;

  char c3{8};
  std::cout << "c3: " << c3 << std::endl;
}
```



There can be confusing while using different versions of GCC compilers. It makes a difference as to which compiler version we are using. With GCC 6.1 and the versions above, we get an error. With the versions below GCC 6.1, we get a warning. To generate an error rather than a warning in the versions below GCC 6.1, we use the `-Werror=narrowing`, flag and the program will generate an error instead. You can try this out with this compiler.

In comparison, the clang++ compiler is much more predictable than GCC.

> 💡 **Compile your program in such a way that narrowing is an error.**

Look at another example below:

direct copy

```
#include <iostream>
using namespace std;

int main() {
  char c3{97};
  std::cout << "c3: " << c3 << std::endl;
}
```

In the code above, if we used the expression `char c3{8}`, it will be indeed not narrow since **8** fits in the type `char`. The same holds for `char c3 = {8}`.

Let's take a look at a couple of examples in the next lesson for a better understanding of this technique.