Unified Initialization with {}

In this lesson, we will learn how to initialize variables using {}.

WE'LL COVER THE FOLLOWING

- Direct initialization
- Copy initialization
- Preventing narrowing

The initialization of variables became uniform in C++11. For unified initialization, we need the `{}` brackets.

> `{}` initialization is always applicable.

Direct initialization

Variables can be declared directly without the assignment operator:

```
std::string str{"my String"};
int i{2011};
```

Copy initialization

`{}` also supports copy initialization with the `=` operator:

```
std::string str = {"my String"};
int i = {2011};
```

The difference is that direct initialization directly calls the constructor of the type, whereas, in copy initialization, the value is created and implicitly converted into the type.

Preventing narrowing

Narrowing, or more precisely narrowing conversion, is an implicit conversion of arithmetic values from one type to another. This can cause a loss of accuracy, which can be extremely dangerous.

The following example shows the issue with the classical way of initializing fundamental types.

The compiler presents a warning, yet the implicit conversions are performed nonetheless, resulting in data loss.

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 two issues:

- The `int` literal, `999`, doesn’t fit into the type `char`.
- The `double` literal, `3.14`, doesn’t fit into the `int` type.

Such an issue is not possible with `{}`-initialization.

> This given code will give an error.

```
#include <iostream>

int main(){
```

```
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;

}
```

Now, the ill-formed program is rejected.

The output also depends on the compiler we use. With **GCC 6.1** and above, we get an error. Any version below that will only produce a warning.

Don't believe me? Try it out with the online compiler https://gcc.godbolt.org/.

The **clang**++ compiler is much more predictable. Therefore, here is a simple tip.

> Compile the program in such a way that narrowing is an error.

We can add the flag, `-Werror=narrowing`, and **GCC 4.8** rejects the program instead of producing a warning.

Let's look at another case:

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

int main() {

  char c1{97};
  char c2 = {97};
  std::cout << "c1: " << c1 << std::endl;
  std::cout << "c2: " << c2 << std::endl;

}
```

The expression `char c1{97}` does not count as narrowing because `97` fits in the `char` type. The same holds true for `char c2 = {97}`.

Let’s look at another example to understand `{}`-initialization better.