- Examples

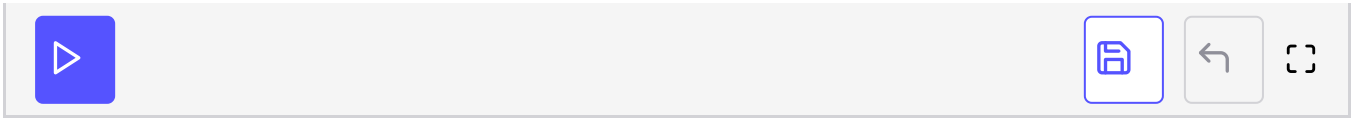
Examples for using {} for uniform initialization and initializer lists.

WE'LL COVER THE FOLLOWING

Example 1

```
// uniformInitialization.cpp

#include <map>
#include <vector>
#include <string>

// Initialization of a C-Array as attribute of a class
class Array{
  public:
    Array(): myData{1,2,3,4,5}{}
  private:
    int myData[5];
};

class MyClass{
  public:
    int x;
    double y;
};

class MyClass2{
  public:
    MyClass2(int fir, double sec):x{fir},y{sec} {};
  private:
    int x;
    double y;
};

int main(){

  // Direct Initialization of a standard container
```

```
  int intArray[] = {1,2,3,4,5};
  std::vector<int> intArray1{1,2,3,4,5};
  std::map<std::string,int> myMap{{"Scott",1976}, {"Dijkstra",1972}};

  // Initialization of a const heap array
  const float* pData= new const float[3]{1.1,2.2,3.3};

  Array arr;

  // Defaut Initialization of a arbitrary object
  int i{};                // i becomes 0
  std::string s{};        // s becomes ""
  std::vector<float> v{}; // v becomes an empty vector
  double d{};             // d becomes 0.0

  // Initializations of an arbitrary object using public attributes
  MyClass myClass{2011,3.14};
  MyClass myClass1 = {2011,3.14};

  // Initializations of an arbitrary object using the constructor
  MyClass2 myClass2{2011,3.14};
  MyClass2 myClass3 = {2011,3.14};

}
```



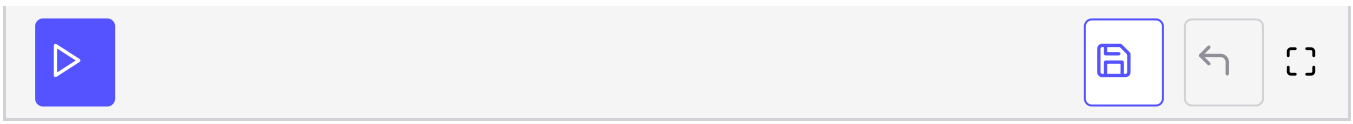
Explanation

- Firstly, the direct initialization of the `C array`, the `std::vector`, and the `std::map` (lines 32 - 34) is quite easy. In the case of the `std::map`, the inner `{}`-pairs are the key and value pairs.
- The next special use case is the direct initialization of a const `C array` on the heap (line 36). The special thing about the array `arr` in line 39 is that `C arrays` can be directly initialized in the constructor initializer (line 10).
- The default initialization in lines 42 - 45 looks quite simple. That does not sound good. Why? Wait for the next section. We directly initialize, in lines 48 and 49, the public attributes of the objects. It is also possible to call the constructor with curly braces (lines 52 and 53).

Example 2

```
// initializerList.cpp

#include <initializer_list>
#include <iostream>
#include <string>
```

```
#include <string>

class MyData{
  public:

    MyData(std::string, int){
      std::cout << "MyData(std::string, int)" << std::endl;
    }

    MyData(int, int){
      std::cout << "MyData(int, int)" << std::endl;
    }

    MyData(std::initializer_list<int>){
      std::cout << "MyData(std::initializer_list<int>)" << std::endl;
    }
};

template<typename T>
void printInitializerList(std::initializer_list<T> inList){
  for (auto& e: inList) std::cout << e << " ";
}

int main(){

  std::cout << std::endl;

  // sequence constructor has a higher priority
  MyData{1, 2};

  // invoke the classical constructor explicitly
  MyData(1, 2);

  // use the classical constructor
  MyData{"dummy", 2};

  std::cout << std::endl;

  // print the initializer list of ints
  printInitializerList({1, 2, 3, 4, 5, 6, 7, 8, 9});

  std::cout << std::endl;

  // print the initializer list of strings
  printInitializerList({"Only", "for", "testing", "purpose."});

  std::cout << "\n\n";
}
```



> **Special Rule:** The `{}` initialization is always applicable. We must remember that if we use automatic type deduction with `auto` in combination with a `{}`-initialization, we will get an `std::initializer_list` in C++14.

Explanation

- When you invoke the constructor with curly braces such as in line 33, the sequence constructor (line 18) is used first. The classical constructor in line 14 serves as a fallback, but this fallback does not work the other way around.
- When we invoke the constructor with round braces such as in line 36, the sequence constructor does no fallback for the classical constructor in line 18. The sequence constructor takes a `std::initalizer_list`.

> 💡 In C++14, `auto` with `{}` always generates `initializer_list`.

```
auto a = {42};   // std::initializer_list<int>
auto b {42};     // std::initializer_list<int>
auto c = {1, 2}; // std::initializer_list<int>
auto d {1, 2};   // std::initializer_list<int>
```

> 💡 With C++17, the rules are more complicated than intuitive.

```
auto a = {42};   // std::initializer_list<int>
auto b {42};     // int
auto c = {1, 2}; // std::initializer_list<int>
auto d {1, 2};   // error, too many
```

You can read the details here.

Let's test your understanding with an exercise in the next lesson.