# - Examples

This lesson presents two more examples of uniform initialization.

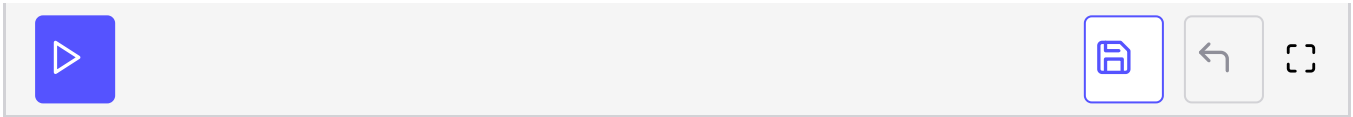
WE'LL COVER THE FOLLOWING



## {} with different types and structures #

```
#include <unordered_map>
#include <string>
#include <vector>

struct MyStruct{
  int x;
  double y;
};

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

struct Telephone{
  std::string name;
  int number;
};

Telephone getTelephone(){
  // Telephone("Rainer Grimm", 12345) created
  return {"Rainer Grimm", 12345};
}

struct MyArray {
    public:
        MyArray(): data {1, 2, 3, 4, 5} {}
    private:
        const int data[5];
};
```

```
void getVector(const std::vector<int>& v){
  // some code

}

int main(){

  // built-in datatypes and strings
  bool b{true};
  bool b2 = {true};
  int i{2011};
  int i2 = {2011};
  std::string s{"string"};
  std::string s2 = {"string"};

  // struct and class
  MyStruct basic{5, 3.2};
  MyStruct basic2 = {5, 3.2};
  MyClass alsoClass{5, 3.2};
  MyClass alsoClass2 = {5, 3.2};

  // C-Array
  // dynamic array initialization
  const float * pData = new const float[4] { 1.5, 4, 3.5, 4.5 };

  // STL-Container
  // a vector of 1 element
  std::vector<int> oneElement{1};
  std::vector<int> oneElement2= {1};

  std::unordered_map<std::string, int> um { {"Dijkstra", 1972}, {"Scott", 1976}, {"Wilkes", 1

  // special cases
  // brace initialization for a std::vector
  getVector({ oneElement[0], 5, 10, 20, 30 });

  // method
  std::vector<int> v {};
  v.insert(v.end(), { 99, 88, -1, 15 });

  // getTelephone returns an initializer list
  Telephone tel(getTelephone());

}
```



## Explanation #

- The code shows us several instances of the `{}`-initializer being used with different entities.
- In line 23, the `getTelephone` function returns an initializer list that can be used in the `Telephone` constructor. The `{}` initialization automatically maps `{"Rainer Grimm", 12345}` to the `Telephone` type. The `getTelephone()`

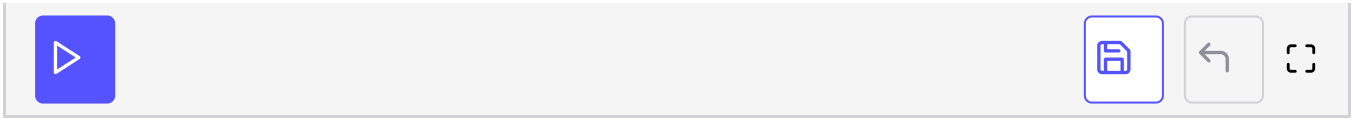
method is used in line 73 to construct a `Telephone` object.

- Line 28 shows another instance of an initializer list for the `data` member of `myArray`.
- Lines 40 to 45 show how variables with basic data types can be initialized using unified initialization.
- Struct and class objects can also be initialized using `{}`, as seen in lines 48 to 51.
- `{}` can also be used to create dynamic arrays, as seen in line 55.
- Lines 59 to 70 show how `{}` works with vectors.

## Initializer lists #

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

class MyData{
  public:

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

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

    MyData(std::initializer_list<int>){         // sequence constructor
      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**: If we use automatic type deduction with `auto` in combination with an {}-initialization, we will get a `std::initializer_list` in C++14.

## Explanation #

- When we invoke the constructor with curly braces such as in line 31, the sequence constructor from line 16 is used first.
- The classical constructor in line 12 serves as a fallback. This fallback does not work the other way around. When we invoke the constructor with round braces, as seen in line 34, the sequence constructor is not a fallback for the classical constructor in line 12.
- The sequence constructor is a constructor that takes an `std::initalizer_list`.

> 💡 In C++14, `auto` with `std::initializer_list` always gives an `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 yet 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
```

We can read more here.

---

Let's test our understanding with an exercise.